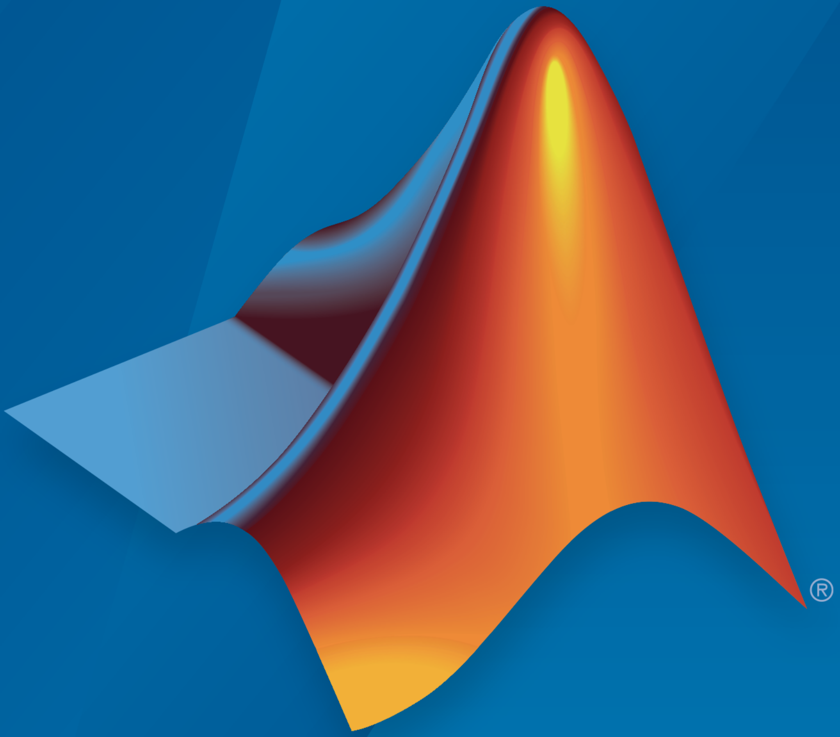
Global Optimization Toolbox

User's Guide



MATLAB®

R2015a

MathWorks®

How to Contact MathWorks

Latest news:	`www.mathworks.com`
Sales and services:	`www.mathworks.com/sales_and_services`
User community:	`www.mathworks.com/matlabcentral`
Technical support:	`www.mathworks.com/support/contact_us`
Phone:	508-647-7000

The MathWorks, Inc.
3 Apple Hill Drive
Natick, MA 01760-2098

Revision History

January 2004	Online only	New for Version 1.0 (Release 13SP1+)
June 2004	First printing	Revised for Version 1.0.1 (Release 14)
October 2004	Online only	Revised for Version 1.0.2 (Release 14SP1)
March 2005	Online only	Revised for Version 1.0.3 (Release 14SP2)
September 2005	Second printing	Revised for Version 2.0 (Release 14SP3)
March 2006	Online only	Revised for Version 2.0.1 (Release 2006a)
September 2006	Online only	Revised for Version 2.0.2 (Release 2006b)
March 2007	Online only	Revised for Version 2.1 (Release 2007a)
September 2007	Third printing	Revised for Version 2.2 (Release 2007b)
March 2008	Online only	Revised for Version 2.3 (Release 2008a)
October 2008	Online only	Revised for Version 2.4 (Release 2008b)
March 2009	Online only	Revised for Version 2.4.1 (Release 2009a)
September 2009	Online only	Revised for Version 2.4.2 (Release 2009b)
March 2010	Online only	Revised for Version 3.0 (Release 2010a)
September 2010	Online only	Revised for Version 3.1 (Release 2010b)
April 2011	Online only	Revised for Version 3.1.1 (Release 2011a)
September 2011	Online only	Revised for Version 3.2 (Release 2011b)
March 2012	Online only	Revised for Version 3.2.1 (Release 2012a)
September 2012	Online only	Revised for Version 3.2.2 (Release 2012b)
March 2013	Online only	Revised for Version 3.2.3 (Release 2013a)
September 2013	Online only	Revised for Version 3.2.4 (Release 2013b)
March 2014	Online only	Revised for Version 3.2.5 (Release 2014a)
October 2014	Online only	Revised for Version 3.3 (Release 2014b)
March 2015	Online only	Revised for Version 3.3.1 (Release 2015a)

Contents

Getting Started

4 Using Direct Search

5 Using the Genetic Algorithm

9 Parallel Processing

10 Options Reference

Getting Started

Introducing Global Optimization Toolbox Functions

Global Optimization Toolbox Product Description

Solve multiple maxima, multiple minima, and nonsmooth optimization problems

Global Optimization Toolbox provides methods that search for global solutions to problems that contain multiple maxima or minima. It includes global search, multistart, pattern search, genetic algorithm, and simulated annealing solvers. You can use these solvers to solve optimization problems where the objective or constraint function is continuous, discontinuous, stochastic, does not possess derivatives, or includes simulations or black-box functions with undefined values for some parameter settings.

Genetic algorithm and pattern search solvers support algorithmic customization. You can create a custom genetic algorithm variant by modifying initial population and fitness scaling options or by defining parent selection, crossover, and mutation functions. You can customize pattern search by defining polling, searching, and other functions.

Key Features

- Interactive tools for defining and solving optimization problems and monitoring solution progress
- Global search and multistart solvers for finding single or multiple global optima
- Genetic algorithm solver that supports linear, nonlinear, and bound constraints
- Multiobjective genetic algorithm with Pareto-front identification, including linear and bound constraints
- Pattern search solver that supports linear, nonlinear, and bound constraints
- Simulated annealing tools that implement a random search method, with options for defining annealing process, temperature schedule, and acceptance criteria
- Parallel computing support in multistart, genetic algorithm, and pattern search solver
- Custom data type support in genetic algorithm, multiobjective genetic algorithm, and simulated annealing solvers

Solution Quality

Global Optimization Toolbox solvers repeatedly attempt to locate a global solution. However, no solver employs an algorithm that can certify a solution as global.

You can extend the capabilities of Global Optimization Toolbox functions by writing your own files, by using them in combination with other toolboxes, or with the MATLAB® or Simulink® environments.

Comparison of Five Solvers

In this section...

Function to Optimize

This example shows how to minimize Rastrigin's function with five solvers. Each solver has its own characteristics. The characteristics lead to different solutions and run times. The results, examined in "Compare Syntax and Solutions" on page 1-10, can help you choose an appropriate solver for your own problems.

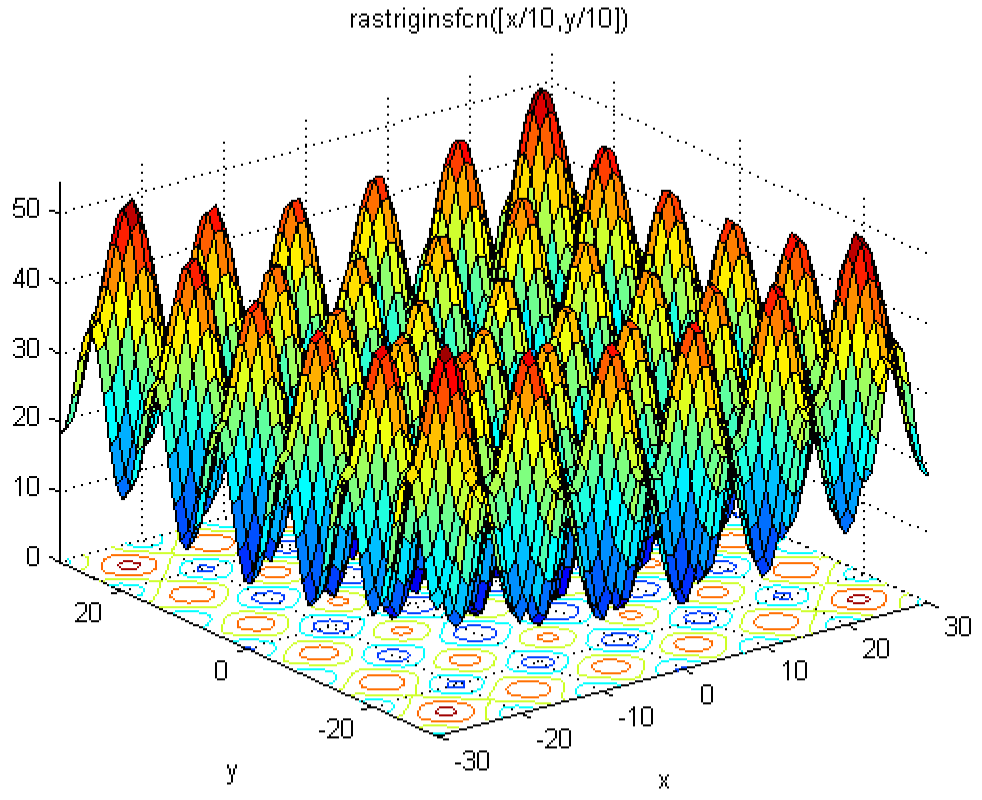
Rastrigin's function has many local minima, with a global minimum at (0,0):

$$\mathrm{Ras}(x) = 20 + x_1^2 + x_2^2 - 10\left(\cos 2\pi x_1 + \cos 2\pi x_2\right).$$

Usually you don't know the location of the global minimum of your objective function. To show how the solvers look for a global solution, this example starts all the solvers around the point `[20,30]`, which is far from the global minimum.

The `rastriginsfcn.m` file implements Rastrigin's function. This file comes with Global Optimization Toolbox software. This example employs a scaled version of Rastrigin's function with larger basins of attraction. For information, see "Basins of Attraction" on page 1-13.

```
rf2 = @(x)rastriginsfcn(x/10);
```



This example minimizes `rf2` using the default settings of `fminunc` (an Optimization Toolbox™ solver), `patternsearch`, and `GlobalSearch`. It also uses `ga` with a nondefault setting, to obtain an initial population around the point `[20,30]`.

Five Solution Methods

- "fminunc" on page 1-6
- "patternsearch" on page 1-6
- "ga" on page 1-7
- "particleswarm" on page 1-8
- "GlobalSearch" on page 1-9

fminunc

To solve the optimization problem using the `fminunc` Optimization Toolbox solver, enter:

```
rf2 = @(x)rastriginsfcn(x/10); % objective
x0 = [20,30]; % start point away from the minimum
[xf,ff,flf,of] = fminunc(rf2,x0)
```

`fminunc` returns

```
Local minimum found.

Optimization completed because the size of the gradient is
less than the default value of the function tolerance.

xf =
   19.8991   29.8486
ff =
   12.9344
flf =
     1
of =
       iterations: 3
        funcCount: 15
         stepsize: 1
    firstorderopt: 5.9907e-09
        algorithm: 'quasi-newton'
          message: 'Local minimum found.

Optimization completed because the size of ...'
```

- `xf` is the minimizing point.
- `ff` is the value of the objective, `rf2`, at `xf`.
- `flf` is the exit flag. An exit flag of `1` indicates `xf` is a local minimum.
- `of` is the output structure, which describes the `fminunc` calculations leading to the solution.

patternsearch

To solve the optimization problem using the `patternsearch` Global Optimization Toolbox solver, enter:

```
rf2 = @(x)rastriginsfcn(x/10); % objective
x0 = [20,30]; % start point away from the minimum
```

```
[xp,fp,flp,op] = patternsearch(rf2,x0)
```

`patternsearch` returns

```
Optimization terminated: mesh size less than options.TolMesh.
xp =
   19.8991   -9.9496
fp =
    4.9748
flp =
     1
op =
        function: @(x)rastriginsfcn(x/10)
     problemtype: 'unconstrained'
      pollmethod: 'gpspositivebasis2n'
    searchmethod: []
      iterations: 48
       funccount: 174
        meshsize: 9.5367e-07
        rngstate: [1x1 struct]
         message: 'Optimization terminated: mesh size less than options.TolMesh.'
```

- `xp` is the minimizing point.
- `fp` is the value of the objective, `rf2`, at `xp`.
- `flp` is the exit flag. An exit flag of `1` indicates `xp` is a local minimum.
- `op` is the output structure, which describes the `patternsearch` calculations leading to the solution.

ga

To solve the optimization problem using the `ga` Global Optimization Toolbox solver, enter:

```
rng default % for reproducibility
rf2 = @(x)rastriginsfcn(x/10); % objective
x0 = [20,30]; % start point away from the minimum
initpop = 10*randn(20,2) + repmat([10 30],20,1);
opts = gaoptimset('InitialPopulation',initpop);
[xga,fga,flga,oga] = ga(rf2,2,[],[],[],[],[],[],[],opts)
```

`initpop` is a 20-by-2 matrix. Each row of `initpop` has mean `[10,30]`, and each element is normally distributed with standard deviation `10`. The rows of `initpop` form an initial population matrix for the `ga` solver.

opts is an optimization structure setting initpop as the initial population.

The final line calls ga, using the optimization structure.

ga uses random numbers, and produces a random result. In this case ga returns:

```
Optimization terminated: average change in the fitness value
less than options.TolFun.
xga =
    0.1191    0.0089
fga =
    0.0283
flga =
     1
oga =
    problemtype: 'unconstrained'
       rngstate: [1x1 struct]
    generations: 107
      funccount: 5400
        message: 'Optimization terminated: average change in the fitness value less...
  maxconstraint: []
```

- xga is the minimizing point.
- fga is the value of the objective, rf2, at xga.
- flga is the exit flag. An exit flag of 1 indicates xga is a local minimum.
- oga is the output structure, which describes the ga calculations leading to the solution.

particleswarm

Like ga, particleswarm is a population-based algorithm. So for a fair comparison of solvers, initialize the particle swarm to the same population as ga.

```
rng default % for reproducibility
rf2 = @(x)rastriginsfcn(x/10); % objective
opts = optimoptions('particleswarm','InitialSwarm',initpop);
[xpso,fpso,flgpso,opso] = particleswarm(rf2,2,[],[],opts)

    Optimization ended: relative change in the objective value
    over the last OPTIONS.StallIterLimit iterations is less than OPTIONS.TolFun.

xpso =
   1.0e-06 *

   -0.8839    0.3073


fpso =

   1.7373e-12


flgpso =

     1


opso =

      rngstate: [1x1 struct]
    iterations: 114
     funccount: 2300
       message: 'Optimization ended: relative change in the objective value 
over the...'
```

- `xpso` is the minimizing point.
- `fpso` is the value of the objective, `rf2`, at `xpso`.
- `flgpso` is the exit flag. An exit flag of `1` indicates `xpso` is a local minimum.
- `opso` is the output structure, which describes the `particleswarm` calculations leading to the solution.

GlobalSearch

To solve the optimization problem using the `GlobalSearch` solver, enter:

```
rf2 = @(x)rastriginsfcn(x/10); % objective
x0 = [20,30]; % start point away from the minimum
problem = createOptimProblem('fmincon','objective',rf2,...
    'x0',x0);
gs = GlobalSearch;
[xg,fg,flg,og] = run(gs,problem)
```

`problem` is an optimization problem structure. `problem` specifies the `fmincon` solver, the `rf2` objective function, and `x0=[20,30]`. For more information on using `createOptimProblem`, see "Create Problem Structure" on page 3-6.

Note: You must specify `fmincon` as the solver for `GlobalSearch`, even for unconstrained problems.

`gs` is a default `GlobalSearch` object. The object contains options for solving the problem. Calling `run(gs,problem)` runs `problem` from multiple start points. The start points are random, so the following result is also random.

In this case, the run returns:

```
GlobalSearch stopped because it analyzed all the trial points.

All 4 local solver runs converged with a positive local solver exit flag.

xg =
   1.0e-07 *
   -0.1405   -0.1405

fg =
     0

flg =
     1

og =
                funcCount: 2178
         localSolverTotal: 4
       localSolverSuccess: 4
    localSolverIncomplete: 0
    localSolverNoSolution: 0
                  message: 'GlobalSearch stopped because it analyzed all the trial po...'
```

- `xg` is the minimizing point.
- `fg` is the value of the objective, `rf2`, at `xg`.
- `flg` is the exit flag. An exit flag of `1` indicates all `fmincon` runs converged properly.
- `og` is the output structure, which describes the `GlobalSearch` calculations leading to the solution.

Compare Syntax and Solutions

One solution is better than another if its objective function value is smaller than the other. The following table summarizes the results, accurate to one decimal.

Results	fminunc	patternsearch	ga	particleswarm	GlobalSearch
solution	`[19.9 29.9]`	`[19.9 -9.9]`	`[0.1 0]`	`[0 0]`	`[0 0]`
objective	12.9	5	0.03	0	0

Results	fminunc	patternsearch	ga	particleswarm	GlobalSearch
# Fevals	15	174	5400	2300	2178

These results are typical:

- `fminunc` quickly reaches the local solution within its starting basin, but does not explore outside this basin at all. `fminunc` has simple calling syntax.
- `patternsearch` takes more function evaluations than `fminunc`, and searches through several basins, arriving at a better solution than `fminunc`. `patternsearch` calling syntax is the same as that of `fminunc`.
- `ga` takes many more function evaluations than `patternsearch`. By chance it arrived at a better solution. `ga` is stochastic, so its results change with every run. `ga` has simple calling syntax, but there are extra steps to have an initial population near `[20,30]`.
- `particleswarm` takes fewer function evaluations than `ga`, but more than `patternsearch`. By chance it arrived at a better solution. In this case, `particleswarm` found the global optimum. `particleswarm` is stochastic, so its results change with every run. `particleswarm` has simple calling syntax, but there are extra steps to have an initial population near `[20,30]`.
- `GlobalSearch` `run` takes the same order of magnitude of function evaluations as `ga` and `particleswarm`, searches many basins, and arrives at a good solution. In this case, `GlobalSearch` found the global optimum. Setting up `GlobalSearch` is more involved than setting up the other solvers. As the example shows, before calling `GlobalSearch`, you must create both a `GlobalSearch` object (`gs` in the example), and a problem structure (`problem`). Then, call the `run` method with `gs` and `problem`. For more details on how to run `GlobalSearch`, see “Optimization Workflow” on page 3-3.

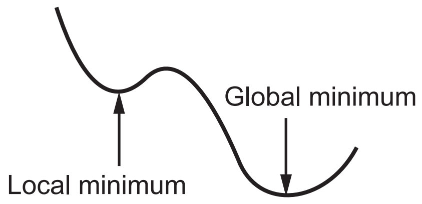
What Is Global Optimization?

In this section...
"Local vs. Global Optima" on page 1-12
"Basins of Attraction" on page 1-13

Local vs. Global Optima

Optimization is the process of finding the point that minimizes a function. More specifically:

- A *local* minimum of a function is a point where the function value is smaller than or equal to the value at nearby points, but possibly greater than at a distant point.
- A *global* minimum is a point where the function value is smaller than or equal to the value at all other feasible points.



Generally, Optimization Toolbox solvers find a local optimum. (This local optimum can be a global optimum.) They find the optimum in the *basin of attraction* of the starting point. For more information, see "Basins of Attraction" on page 1-13.

In contrast, Global Optimization Toolbox solvers are designed to search through more than one basin of attraction. They search in various ways:

- `GlobalSearch` and `MultiStart` generate a number of starting points. They then use a local solver to find the optima in the basins of attraction of the starting points.
- `ga` uses a set of starting points (called the population) and iteratively generates better points from the population. As long as the initial population covers several basins, `ga` can examine several basins.
- `simulannealbnd` performs a random search. Generally, `simulannealbnd` accepts a point if it is better than the previous point. `simulannealbnd` occasionally accepts a worse point, in order to reach a different basin.

- `patternsearch` looks at a number of neighboring points before accepting one of them. If some neighboring points belong to different basins, `patternsearch` in essence looks in a number of basins at once.

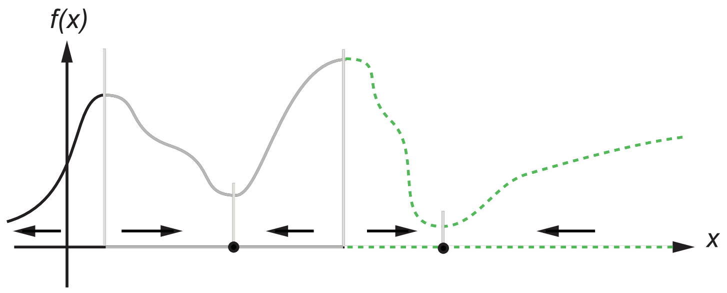
Basins of Attraction

If an objective function $f(x)$ is smooth, the vector $-\nabla f(x)$ points in the direction where $f(x)$ decreases most quickly. The equation of steepest descent, namely

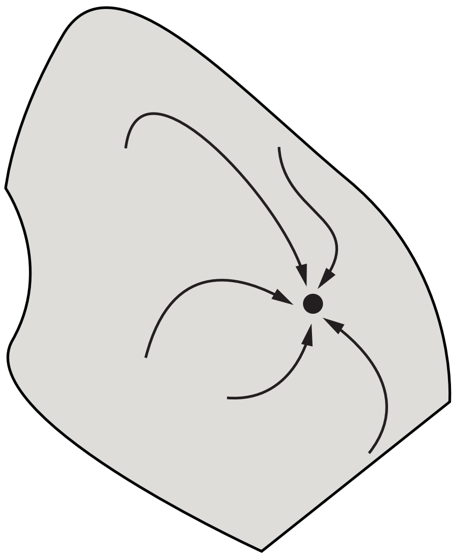
$$\frac{d}{dt}x(t) = -\nabla f(x(t)),$$

yields a path $x(t)$ that goes to a local minimum as t gets large. Generally, initial values $x(0)$ that are close to each other give steepest descent paths that tend to the same minimum point. The *basin of attraction* for steepest descent is the set of initial values leading to the same local minimum.

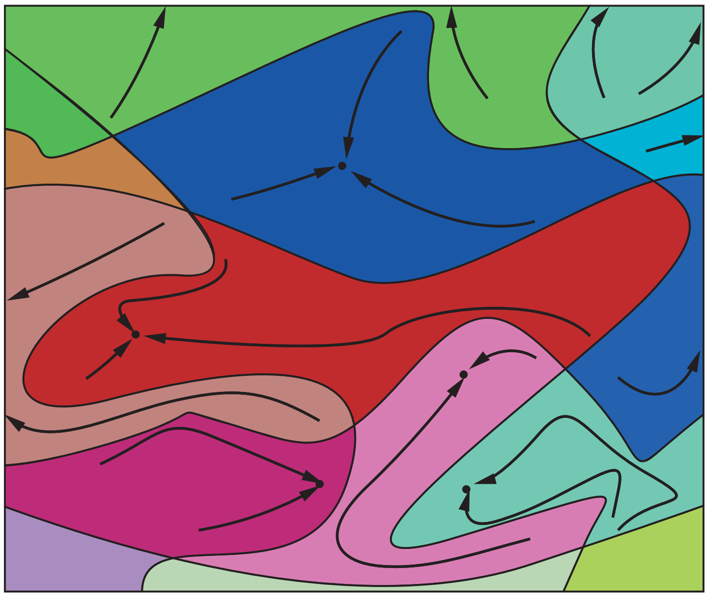
The following figure shows two one-dimensional minima. The figure shows different basins of attraction with different line styles, and it shows directions of steepest descent with arrows. For this and subsequent figures, black dots represent local minima. Every steepest descent path, starting at a point $x(0)$, goes to the black dot in the basin containing $x(0)$.



The following figure shows how steepest descent paths can be more complicated in more dimensions.



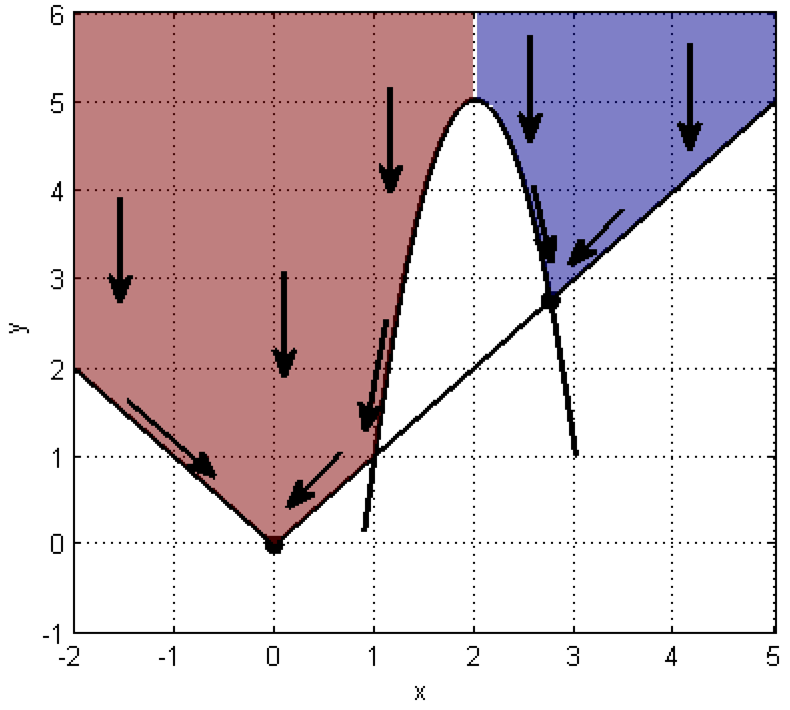
The following figure shows even more complicated paths and basins of attraction.



Constraints can break up one basin of attraction into several pieces. For example, consider minimizing y subject to:

- $y \geq |x|$
- $y \geq 5 - 4(x-2)^2$.

The figure shows the two basins of attraction with the final points.



The steepest descent paths are straight lines down to the constraint boundaries. From the constraint boundaries, the steepest descent paths travel down along the boundaries. The final point is either (0,0) or (11/4,11/4), depending on whether the initial x-value is above or below 2.

Optimization Workflow

To solve an optimization problem:

1. Decide what type of problem you have, and whether you want a local or global solution (see “Local vs. Global Optima” on page 1-12). Choose a solver per the recommendations in “Choosing a Solver” on page 1-17.
2. Write your objective function and, if applicable, constraint functions per the syntax in “Compute Objective Functions” on page 2-2 and “Write Constraints” on page 2-6.
3. Set appropriate options with `psoptimset`, `gaoptimset`, or `saoptimset`, or prepare a `GlobalSearch` or `MultiStart` problem as described in “Optimization Workflow” on page 3-3. For details, see “Pattern Search Options” on page 10-9, “Genetic Algorithm Options” on page 10-28, or “Simulated Annealing Options” on page 10-62.
4. Run the solver.
5. Examine the result. For information on the result, see “Solver Outputs and Iterative Display” in the Optimization Toolbox documentation or Examine Results for `GlobalSearch` or `MultiStart`.
6. If the result is unsatisfactory, change options or start points or otherwise update your optimization and rerun it. For information, see Improve Results, or see “When the Solver Fails ”, “When the Solver Might Have Succeeded ”, or “When the Solver Succeeds ” in the Optimization Toolbox documentation.

Choosing a Solver

In this section...
"Table for Choosing a Solver" on page 1-17
"Solver Characteristics" on page 1-21
"Why Are Some Solvers Objects?" on page 1-23

Table for Choosing a Solver

There are seven Global Optimization Toolbox solvers:

- `ga` (genetic algorithm)
- `GlobalSearch`
- `MultiStart`
- `patternsearch`, also called direct search
- `particleswarm`
- `simulannealbnd` (simulated annealing)
- `gamultiobj`, which is not a minimizer; see "Multiobjective Optimization"

Choose an optimizer based on problem characteristics and on the type of solution you want. "Solver Characteristics" on page 1-21 contains more information to help you decide which solver is likely to be most suitable.

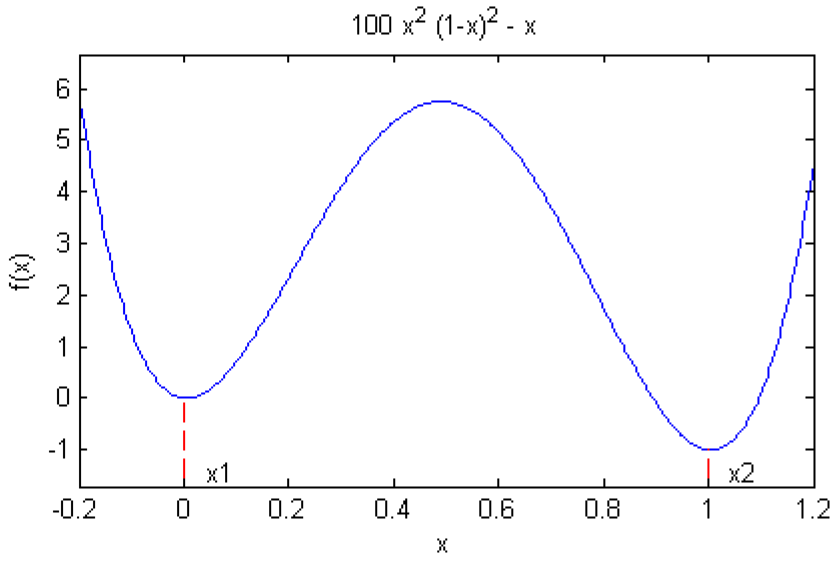
Desired Solution	**Smooth Objective and Constraints**	**Nonsmooth Objective or Constraints**
"Explanation of "Desired Solution"" on page 1-18	"Choosing Between Solvers for Smooth Problems" on page 1-19	"Choosing Between Solvers for Nonsmooth Problems" on page 1-20
Single local solution	Optimization Toolbox functions; see "Optimization Decision Table" in the Optimization Toolbox documentation	`fminbnd`, `patternsearch`, `fminsearch`, `ga`, `particleswarm`, `simulannealbnd`
Multiple local solutions	`GlobalSearch`, `MultiStart`	
Single global solution	`GlobalSearch`, `MultiStart`, `patternsearch`, `ga`, `simulannealbnd`	`patternsearch`, `ga`, `particleswarm`, `simulannealbnd`

Desired Solution	Smooth Objective and Constraints	Nonsmooth Objective or Constraints
Single local solution using parallel processing	`MultiStart`, Optimization Toolbox functions	`patternsearch`, `ga`, `particleswarm`
Multiple local solutions using parallel processing	`MultiStart`	
Single global solution using parallel processing	`MultiStart`	`patternsearch`, `ga`, `particleswarm`

Explanation of "Desired Solution"

To understand the meaning of the terms in "Desired Solution," consider the example $f(x)=100x^2(1–x)^2–x$,

which has local minima `x1` near 0 and `x2` near 1:



The minima are located at:

```
x1 = fminsearch(@(x)(100*x^2*(x - 1)^2 - x),0)
x1 =
    0.0051

x2 = fminsearch(@(x)(100*x^2*(x - 1)^2 - x),1)
```

```
x2 =
    1.0049
```

Description of the Terms

Term	Meaning
Single local solution	Find one local solution, a point *x* where the objective function *f*(*x*) is a local minimum. For more details, see "Local vs. Global Optima" on page 1-12. In the example, both `x1` and `x2` are local solutions.
Multiple local solutions	Find a set of local solutions. In the example, the complete set of local solutions is `{x1,x2}`.
Single global solution	Find the point *x* where the objective function *f*(*x*) is a global minimum. In the example, the global solution is `x2`.

Choosing Between Solvers for Smooth Problems

Single Global Solution

1. Try `GlobalSearch` first. It is most focused on finding a global solution, and has an efficient local solver, `fmincon`.
2. Try `MultiStart` second. It has efficient local solvers, and can search a wide variety of start points.
3. Try `patternsearch` third. It is less efficient, since it does not use gradients. However, `patternsearch` is robust and is more efficient than the remaining local solvers.
4. Try `particleswarm` fourth, if your problem is unconstrained or has only bound constraints. Usually, `particleswarm` is more efficient than the remaining solvers, and can be more efficient than `patternsearch`.
5. Try `ga` fifth. It can handle all types of constraints, and is usually more efficient than `simulannealbnd`.
6. Try `simulannealbnd` last. It can handle problems with no constraints or bound constraints. `simulannealbnd` is usually the least efficient solver. However, given a slow enough cooling schedule, it can find a global solution.

Multiple Local Solutions

`GlobalSearch` and `MultiStart` both provide multiple local solutions. For the syntax to obtain multiple solutions, see "Multiple Solutions" on page 3-25. `GlobalSearch` and `MultiStart` differ in the following characteristics:

- `MultiStart` can find more local minima. This is because `GlobalSearch` rejects many generated start points (initial points for local solution). Essentially, `GlobalSearch` accepts a start point only when it determines that the point has a good chance of obtaining a global minimum. In contrast, `MultiStart` passes all generated start points to a local solver. For more information, see "GlobalSearch Algorithm" on page 3-46.
- `MultiStart` offers a choice of local solver: `fmincon`, `fminunc`, `lsqcurvefit`, or `lsqnonlin`. The `GlobalSearch` solver uses only `fmincon` as its local solver.
- `GlobalSearch` uses a scatter-search algorithm for generating start points. In contrast, `MultiStart` generates points uniformly at random within bounds, or allows you to provide your own points.
- `MultiStart` can run in parallel. See "How to Use Parallel Processing" on page 9-12.

Choosing Between Solvers for Nonsmooth Problems

Choose the applicable solver with the lowest number. For problems with integer constraints, use `ga`.

1. Use `fminbnd` first on one-dimensional bounded problems only. `fminbnd` provably converges quickly in one dimension.
2. Use `patternsearch` on any other type of problem. `patternsearch` provably converges, and handles all types of constraints.
3. Try `fminsearch` next for low-dimensional unbounded problems. `fminsearch` is not as general as `patternsearch` and can fail to converge. For low-dimensional problems, `fminsearch` is simple to use, since it has few tuning options.
4. Try `particleswarm` next on unbounded or bound-constrained problems. `particleswarm` has little supporting theory, but is often an efficient algorithm.
5. Try `ga` next. `ga` has little supporting theory and is often less efficient than `patternsearch` or `particleswarm`. It handles all types of constraints. `ga` is the only solver that handles integer constraints.
6. Try `simulannealbnd` last for unbounded problems, or for problems with bounds. `simulannealbnd` provably converges only for a logarithmic cooling schedule, which is extremely slow. `simulannealbnd` takes only bound constraints, and is often less efficient than `ga`.

Solver Characteristics

Solver	Convergence	Characteristics
`GlobalSearch`	Fast convergence to local optima for smooth problems.	Deterministic iterates
		Gradient-based
		Automatic stochastic start points
		Removes many start points heuristically
`MultiStart`	Fast convergence to local optima for smooth problems.	Deterministic iterates
		Can run in parallel; see “How to Use Parallel Processing” on page 9-12
		Gradient-based
		Stochastic or deterministic start points, or combination of both
		Automatic stochastic start points
		Runs all start points
		Choice of local solver: `fmincon`, `fminunc`, `lsqcurvefit`, or `lsqnonlin`
`patternsearch`	Proven convergence to local optimum, slower than gradient-based solvers.	Deterministic iterates
		Can run in parallel; see “How to Use Parallel Processing” on page 9-12
		No gradients
		User-supplied start point
`particleswarm`	No convergence proof.	Stochastic iterates
		Can run in parallel; see “How to Use Parallel Processing” on page 9-12
		Population-based
		No gradients
		Automatic start population, or user-supplied population, or combination of both

Solver	Convergence	Characteristics
		Only bound constraints
`ga`	No convergence proof.	Stochastic iterates
		Can run in parallel; see "How to Use Parallel Processing" on page 9-12
		Population-based
		No gradients
		Allows integer constraints; see "Mixed Integer Optimization" on page 5-27
		Automatic start population, or user-supplied population, or combination of both
`simulannealbnd`	Proven to converge to global optimum for bounded problems with very slow cooling schedule.	Stochastic iterates
		No gradients
		User-supplied start point
		Only bound constraints

Explanation of some characteristics:

- Convergence — Solvers can fail to converge to any solution when started far from a local minimum. When started near a local minimum, gradient-based solvers converge to a local minimum quickly for smooth problems. `patternsearch` provably converges for a wide range of problems, but the convergence is slower than gradient-based solvers. Both `ga` and `simulannealbnd` can fail to converge in a reasonable amount of time for some problems, although they are often effective.
- Iterates — Solvers iterate to find solutions. The steps in the iteration are iterates. Some solvers have deterministic iterates. Others use random numbers and have stochastic iterates.
- Gradients — Some solvers use estimated or user-supplied derivatives in calculating the iterates. Other solvers do not use or estimate derivatives, but use only objective and constraint function values.
- Start points — Most solvers require you to provide a starting point for the optimization. One reason they require a start point is to obtain the dimension of the decision variables. `ga` does not require any starting points, because it takes the

dimension of the decision variables as an input. `ga` can generate its start population automatically.

Compare the characteristics of Global Optimization Toolbox solvers to Optimization Toolbox solvers.

Solver	Convergence	Characteristics
`fmincon`, `fminunc`, `fseminf`, `lsqcurvefit`, `lsqnonlin`	Proven quadratic convergence to local optima for smooth problems	Deterministic iterates
		Gradient-based
		User-supplied starting point
`fminsearch`	No convergence proof — counterexamples exist.	Deterministic iterates
		No gradients
		User-supplied start point
		No constraints
`fminbnd`	Proven convergence to local optima for smooth problems, slower than quadratic.	Deterministic iterates
		No gradients
		User-supplied start point
		Only one-dimensional problems

All these Optimization Toolbox solvers:

- Have deterministic iterates
- Start from one user-supplied point
- Search just one basin of attraction

Why Are Some Solvers Objects?

`GlobalSearch` and `MultiStart` are objects. What does this mean for you?

- You create a `GlobalSearch` or `MultiStart` object before running your problem.
- You can reuse the object for running multiple problems.
- `GlobalSearch` and `MultiStart` objects are containers for algorithms and global options. You use these objects to run a local solver multiple times. The local solver has its own options.

For more information, see the “Object-Oriented Programming” documentation.

Write Files for Optimization Functions

Compute Objective Functions

In this section...

Objective (Fitness) Functions

To use Global Optimization Toolbox functions, first write a file (or an anonymous function) that computes the function you want to optimize. This is called an objective function for most solvers, or fitness function for `ga`. The function should accept a vector, whose length is the number of independent variables, and return a scalar. For `gamultiobj`, the function should return a row vector of objective function values. For vectorized solvers, the function should accept a matrix, where each row represents one input vector, and return a vector of objective function values. This section shows how to write the file.

Write a Function File

This example shows how to write a file for the function you want to optimize. Suppose that you want to minimize the function

$$f(x) = x_1^2 - 2x_1x_2 + 6x_1 + 4x_2^2 - 3x_2.$$

The file that computes this function must accept a vector `x` of length 2, corresponding to the variables x_1 and x_2, and return a scalar equal to the value of the function at `x`.

1 Select **New > Script** (**Ctrl+N**) from the MATLAB **File** menu. A new file opens in the editor.
2 Enter the following two lines of code:

```
function z = my_fun(x)
```

```
    z = x(1)^2 - 2*x(1)*x(2) + 6*x(1) + 4*x(2)^2 - 3*x(2);
```

3 Save the file in a folder on the MATLAB path.

Check that the file returns the correct value.

```
my_fun([2 3])

ans =
    31
```

For `gamultiobj`, suppose you have three objectives. Your objective function returns a three-element vector consisting of the three objective function values:

```
function z = my_fun(x)
z = zeros(1,3); % allocate output
z(1) = x(1)^2 - 2*x(1)*x(2) + 6*x(1) + 4*x(2)^2 - 3*x(2);
z(2) = x(1)*x(2) + cos(3*x(2)/(2+x(1)));
z(3) = tanh(x(1) + x(2));
```

Write a Vectorized Function

The `ga`, `gamultiobj`, and `patternsearch` solvers optionally compute the objective functions of a collection of vectors in one function call. This method can take less time than computing the objective functions of the vectors serially. This method is called a vectorized function call.

To compute in vectorized fashion:

- Write your objective function to:
  - Accept a matrix with an arbitrary number of rows.
  - Return the vector of function values of each row.
  - For `gamultiobj`, return a matrix, where each row contains the objective function values of the corresponding input matrix row.
- If you have a nonlinear constraint, be sure to write the constraint in a vectorized fashion. For details, see “Vectorized Constraints” on page 2-7.
- Set the `Vectorized` option to `'on'` with `gaoptimset` or `psoptimset`, or set **User function evaluation > Evaluate objective/fitness and constraint functions** to `vectorized` in the Optimization app. For `patternsearch`, also set `CompletePoll` to `'on'`. Be sure to pass the options structure to the solver.

For example, to write the objective function of “Write a Function File” on page 2-2 in a vectorized fashion,

```
function z = my_fun(x)
z = x(:,1).^2 - 2*x(:,1).*x(:,2) + 6*x(:,1) + ...
   4*x(:,2).^2 - 3*x(:,2);
```

To use `my_fun` as a vectorized objective function for `patternsearch`:

```
options = psoptimset('CompletePoll','on','Vectorized','on');
[x fval] = patternsearch(@my_fun,[1 1],[],[],[],[],[],[],...
    [],options);
```

To use `my_fun` as a vectorized objective function for `ga`:

```
options = gaoptimset('Vectorized','on');
[x fval] = ga(@my_fun,2,[],[],[],[],[],[],[],options);
```

For `gamultiobj`,

```
function z = my_fun(x)
z = zeros(size(x,1),3); % allocate output
z(:,1) = x(:,1).^2 - 2*x(:,1).*x(:,2) + 6*x(:,1) + ...
   4*x(:,2).^2 - 3*x(:,2);
z(:,2) = x(:,1).*x(:,2) + cos(3*x(:,2)./(2+x(:,1)));
z(:,3) = tanh(x(:,1) + x(:,2));
```

To use `my_fun` as a vectorized objective function for `gamultiobj`:

```
options = gaoptimset('Vectorized','on');
[x fval] = gamultiobj(@my_fun,2,[],[],[],[],[],[],options);
```

For more information on writing vectorized functions for `patternsearch`, see “Vectorize the Objective and Constraint Functions” on page 4-80. For more information on writing vectorized functions for `ga`, see “Vectorize the Fitness Function” on page 5-110.

Gradients and Hessians

If you use `GlobalSearch` or `MultiStart`, your objective function can return derivatives (gradient, Jacobian, or Hessian). For details on how to include this syntax in your objective function, see “Writing Objective Functions” in the Optimization Toolbox documentation. Use `optimoptions` to set options so that your solver uses the derivative information:

Local Solver = fmincon, fminunc

Condition	Option Setting
Objective function contains gradient	`'GradObj' = 'on'`
Objective function contains Hessian	`'Hessian' = 'on'`
Constraint function contains gradient	`'GradConstr' = 'on'`
Calculate Hessians of Lagrangian in an extra function	`'Hessian' = 'on'`, `'HessFcn' =` function handle

For more information about Hessians for `fmincon`, see “Hessian”.

Local Solver = lsqcurvefit, lsqnonlin

Condition	Option Setting
Objective function contains Jacobian	`'Jacobian' = 'on'`

Maximizing vs. Minimizing

Global Optimization Toolbox optimization functions minimize the objective or fitness function. That is, they solve problems of the form

$$\min_x f(x).$$

If you want to maximize $f(x)$, minimize $–f(x)$, because the point at which the minimum of $–f(x)$ occurs is the same as the point at which the maximum of $f(x)$ occurs.

For example, suppose you want to maximize the function

$$f(x) = x_1^2 - 2x_1x_2 + 6x_1 + 4x_2^2 - 3x_2.$$

Write your function file to compute

$$g(x) = -f(x) = -x_1^2 + 2x_1x_2 - 6x_1 - 4x_2^2 + 3x_2,$$

and minimize $g(x)$.

Write Constraints

In this section...

Consult Optimization Toolbox Documentation

Many Global Optimization Toolbox functions accept bounds, linear constraints, or nonlinear constraints. To see how to include these constraints in your problem, see "Writing Constraints" in the Optimization Toolbox documentation. Try consulting these pertinent links to sections:

- "Bound Constraints"
- "Linear Inequality Constraints" (linear equality constraints have the same form)
- "Nonlinear Constraints "

Set Bounds

It is more important to set bounds for global solvers than for local solvers. Global solvers use bounds in a variety of ways:

- `GlobalSearch` requires bounds for its scatter-search point generation. If you do not provide bounds, `GlobalSearch` bounds each component below by `-9999` and above by `10001`. However, these bounds can easily be inappropriate.
- If you do not provide bounds and do not provide custom start points, `MultiStart` bounds each component below by `-1000` and above by `1000`. However, these bounds can easily be inappropriate.
- `ga` uses bounds and linear constraints for its initial population generation. For unbounded problems, `ga` uses a default of `0` as the lower bound and `1` as the upper bound for each dimension for initial point generation. For bounded problems, and problems with linear constraints, `ga` uses the bounds and constraints to make the initial population.

- simulannealbnd and patternsearch do not require bounds, although they can use bounds.

Ensure ga Options Maintain Feasibility

The ga solver generally maintains strict feasibility with respect to bounds and linear constraints. This means that, at every iteration, all members of a population satisfy the bounds and linear constraints.

However, you can set options that cause this feasibility to fail. For example if you set MutationFcn to @mutationgaussian or @mutationuniform, the mutation function does not respect constraints, and your population can become infeasible. Similarly, some crossover functions can cause infeasible populations, although the default gacreationlinearfeasible does respect bounds and linear constraints. Also, ga can have infeasible points when using custom mutation or crossover functions.

To ensure feasibility, use the default crossover and mutation functions for ga. Be especially careful that any custom functions maintain feasibility with respect to bounds and linear constraints.

Gradients and Hessians

If you use GlobalSearch or MultiStart with fmincon, your nonlinear constraint functions can return derivatives (gradient or Hessian). For details, see “Gradients and Hessians” on page 2-4.

Vectorized Constraints

The ga and patternsearch solvers optionally compute the nonlinear constraint functions of a collection of vectors in one function call. This method can take less time than computing the objective functions of the vectors serially. This method is called a vectorized function call.

For the solver to compute in a vectorized manner, you must vectorize both your objective (fitness) function and nonlinear constraint function. For details, see “Vectorize the Objective and Constraint Functions” on page 4-80.

As an example, suppose your nonlinear constraints for a three-dimensional problem are

$$\begin{aligned}\frac{x_1^2}{4}+\frac{x_2^2}{9}+\frac{x_3^2}{25} &\le 6\\ x_3 &\ge \cosh(x_1+x_2)\\ x_1x_2x_3 &= 2.\end{aligned}$$

The following code gives these nonlinear constraints in a vectorized fashion, assuming that the rows of your input matrix `x` are your population or input vectors:

```
function [c ceq] = nlinconst(x)

c(:,1) = x(:,1).^2/4 + x(:,2).^2/9 + x(:,3).^2/25 - 6;
c(:,2) = cosh(x(:,1) + x(:,2)) - x(:,3);
ceq = x(:,1).*x(:,2).*x(:,3) - 2;
```

For example, minimize the vectorized quadratic function

```
function y = vfun(x)
y = -x(:,1).^2 - x(:,2).^2 - x(:,3).^2;
```

over the region with constraints `nlinconst` using `patternsearch`:

```
options = psoptimset('CompletePoll','on','Vectorized','on');
[x fval] = patternsearch(@vfun,[1,1,2],[],[],[],[],[],[],...
    @nlinconst,options)
Optimization terminated: mesh size less than options.TolMesh
 and constraint violation is less than options.TolCon.

x =
    0.2191    0.7500   12.1712

fval =
 -148.7480
```

Using `ga`:

```
options = gaoptimset('Vectorized','on');
[x fval] = ga(@vfun,3,[],[],[],[],[],[],@nlinconst,options)
Optimization terminated: maximum number of generations exceeded.

x =
   -1.4098   -0.1216   11.6664

fval =
 -138.1066
```

For this problem `patternsearch` computes the solution far more quickly and accurately.

Using GlobalSearch and MultiStart

Problems That GlobalSearch and MultiStart Can Solve

The `GlobalSearch` and `MultiStart` solvers apply to problems with smooth objective and constraint functions. The solvers search for a global minimum, or for a set of local minima. For more information on which solver to use, see "Choosing a Solver" on page 1-17.

`GlobalSearch` and `MultiStart` work by starting a local solver, such as `fmincon`, from a variety of start points. Generally the start points are random. However, for `MultiStart` you can provide a set of start points. For more information, see "How GlobalSearch and MultiStart Work" on page 3-44.

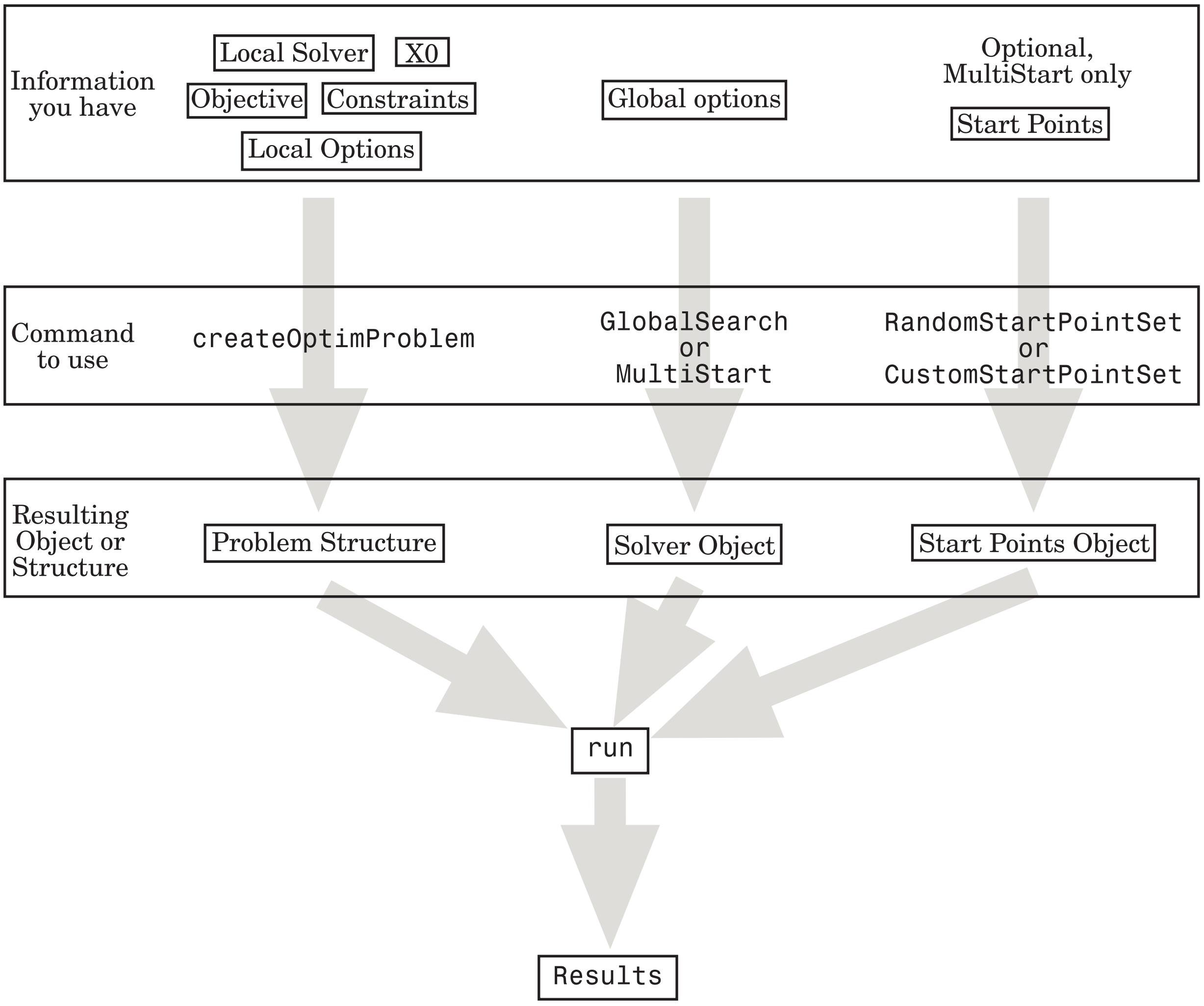
To find out how to use these solvers, see "Optimization Workflow" on page 3-3.

Optimization Workflow

To find a global or multiple local solutions:

1. "Create Problem Structure" on page 3-6
2. "Create Solver Object" on page 3-14
3. (Optional, `MultiStart` only) "Set Start Points for MultiStart" on page 3-17
4. "Run the Solver" on page 3-20

The following figure illustrates these steps.


Information you have
Local Solver
X0
Objective
Constraints
Local Options
Global options
Optional, MultiStart only
Start Points
Command to use
createOptimProblem
GlobalSearch or MultiStart
RandomStartPointSet or CustomStartPointSet
Resulting Object or Structure
Problem Structure
Solver Object
Start Points Object
run
Results

Inputs for Problem Structure

A problem structure defines a local optimization problem using a local solver, such as `fmincon`. Therefore, most of the documentation on choosing a solver or preparing the inputs is in the Optimization Toolbox documentation.

Required Inputs

Input	More Information
Local solver	"Optimization Decision Table" in the Optimization Toolbox documentation.
Objective function	"Compute Objective Functions" on page 2-2
Start point `x0`	Gives the dimension of points for the objective function.

Optional Inputs

Input	More Information
Constraint functions	"Write Constraints" on page 2-6
Local options structure	"Set Options"

Create Problem Structure

In this section...
"About Problem Structures" on page 3-6
"Using the createOptimProblem Function" on page 3-6
"Exporting from the Optimization app" on page 3-9

About Problem Structures

To use the `GlobalSearch` or `MultiStart` solvers, you must first create a problem structure. There are two ways to create a problem structure: using the `createOptimProblem` function and exporting from the Optimization app.

For information on creating inputs for the problem structure, see "Inputs for Problem Structure" on page 3-5.

Using the createOptimProblem Function

Follow these steps to create a problem structure using the `createOptimProblem` function.

1. Define your objective function as a file or anonymous function. For details, see "Compute Objective Functions" on page 2-2. If your solver is `lsqcurvefit` or `lsqnonlin`, ensure the objective function returns a vector, not scalar.
2. If relevant, create your constraints, such as bounds and nonlinear constraint functions. For details, see "Write Constraints" on page 2-6.
3. Create a start point. For example, to create a three-dimensional random start point `xstart`:

   ```
   xstart = randn(3,1);
   ```
4. (Optional) Create an options structure using `optimoptions`. For example,

   ```
   options = optimoptions(@fmincon,'Algorithm','interior-point');
   ```
5. Enter

   ```
   problem = createOptimProblem(solver,
   ```

where *`solver`* is the name of your local solver:

- For `GlobalSearch`: `'fmincon'`
- For `MultiStart` the choices are:
  - `'fmincon'`
  - `'fminunc'`
  - `'lsqcurvefit'`
  - `'lsqnonlin'`

  For help choosing, see "Optimization Decision Table".

6 Set an initial point using the `'x0'` parameter. If your initial point is `xstart`, and your solver is `fmincon`, your entry is now

```
problem = createOptimProblem('fmincon','x0',xstart,
```

7 Include the function handle for your objective function in `objective`:

```
problem = createOptimProblem('fmincon','x0',xstart, ...
    'objective',@objfun,
```

8 Set bounds and other constraints as applicable.

Constraint	Name
lower bounds	`'lb'`
upper bounds	`'ub'`
matrix `Aineq` for linear inequalities `Aineq x` ≤ `bineq`	`'Aineq'`
vector `bineq` for linear inequalities `Aineq x` ≤ `bineq`	`'bineq'`
matrix `Aeq` for linear equalities `Aeq x = beq`	`'Aeq'`
vector `beq` for linear equalities `Aeq x = beq`	`'beq'`
nonlinear constraint function	`'nonlcon'`

9 If using the `lsqcurvefit` local solver, include vectors of input data and response data, named `'xdata'` and `'ydata'` respectively.

10 *Best practice: validate the problem structure by running your solver on the structure.* For example, if your local solver is `fmincon`:

```
[x fval eflag output] = fmincon(problem);
```

Example: Creating a Problem Structure with createOptimProblem

This example minimizes the function from "Run the Solver" on page 3-20, subject to the constraint $x_1 + 2x_2 \geq 4$. The objective is
sixmin $= 4x^2 - 2.1x^4 + x^6/3 + xy - 4y^2 + 4y^4$.

Use the `interior-point` algorithm of `fmincon`, and set the start point to `[2;3]`.

1 Write a function handle for the objective function.

```
sixmin = @(x)(4*x(1)^2 - 2.1*x(1)^4 + x(1)^6/3 ...
    + x(1)*x(2) - 4*x(2)^2 + 4*x(2)^4);
```

2 Write the linear constraint matrices. Change the constraint to "less than" form:

```
A = [-1,-2];
b = -4;
```

3 Create the local options structure to use the `interior-point` algorithm:

```
opts = optimoptions(@fmincon,'Algorithm','interior-point');
```

4 Create the problem structure with `createOptimProblem`:

```
problem = createOptimProblem('fmincon', ...
    'x0',[2;3],'objective',sixmin, ...
    'Aineq',A,'bineq',b,'options',opts)
```

5 The resulting structure:

```
problem =
    objective: @(x)(4*x(1)^2-2.1*x(1)^4+x(1)^6/3+x(1)*x(2)-4*x(2)^2+4*x(2)^4)
           x0: [2x1 double]
        Aineq: [-1 -2]
        bineq: -4
          Aeq: []
          beq: []
           lb: []
           ub: []
      nonlcon: []
       solver: 'fmincon'
      options: [1x1 optim.options.Fmincon]
```

6 *Best practice: validate the problem structure by running your solver on the structure:*

```
[x fval eflag output] = fmincon(problem);
```

Exporting from the Optimization app

Follow these steps to create a problem structure using the Optimization app.

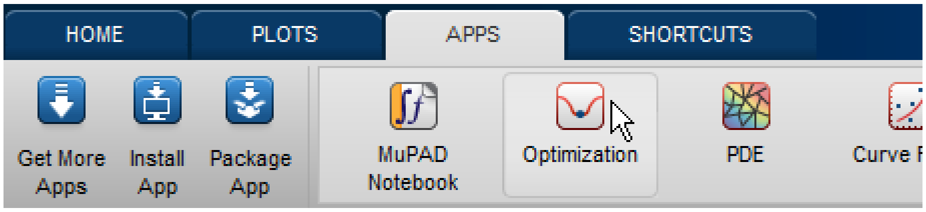
1 Define your objective function as a file or anonymous function. For details, see "Compute Objective Functions" on page 2-2. If your solver is `lsqcurvefit` or `lsqnonlin`, ensure the objective function returns a vector, not scalar.

2 If relevant, create nonlinear constraint functions. For details, see "Nonlinear Constraints ".

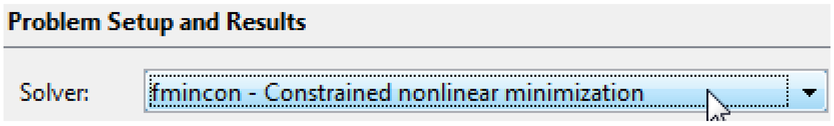
3 Create a start point. For example, to create a three-dimensional random start point `xstart`:

```
xstart = randn(3,1);
```

4 Open the Optimization app by entering `optimtool` at the command line, or by choosing the Optimization app from the **Apps** tab.



5 Choose the local **Solver**.

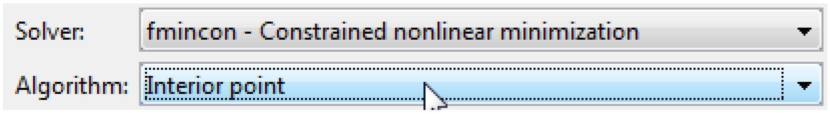


Solver: fmincon - Constrained nonlinear minimization

- For `GlobalSearch`: `fmincon` (default).
- For `MultiStart`:
  - `fmincon` (default)
  - `fminunc`
  - `lsqcurvefit`
  - `lsqnonlin`

For help choosing, see "Optimization Decision Table".

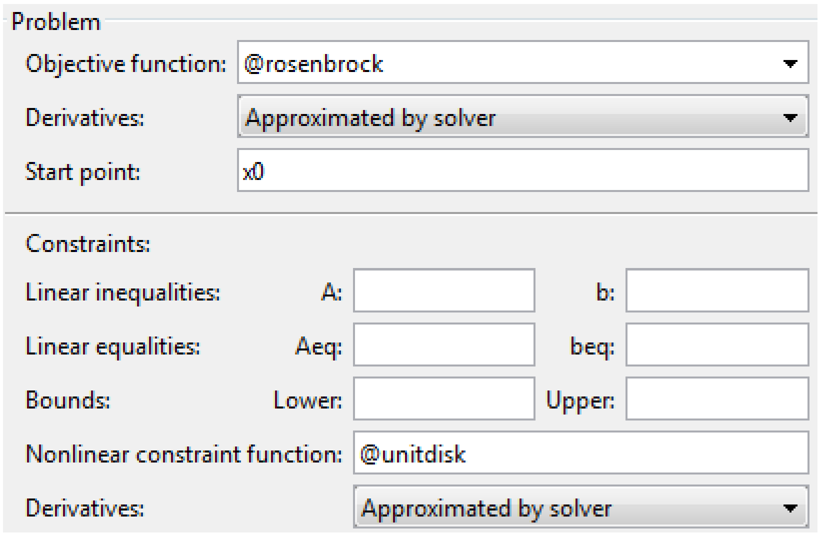
6 Choose an appropriate **Algorithm**. For help choosing, see “Choosing the Algorithm”.


Solver: fmincon - Constrained nonlinear minimization

Algorithm: Interior point

7 Set an initial point (**Start point**).

8 Include the function handle for your objective function in **Objective function**, and, if applicable, include your **Nonlinear constraint function**. For example,


Problem

Objective function: @rosenbrock

Derivatives: Approximated by solver

Start point: x0

Constraints:

Linear inequalities: A: b:

Linear equalities: Aeq: beq:

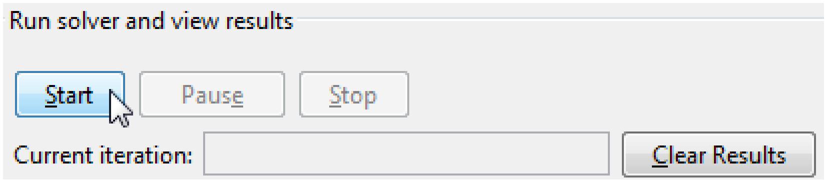
Bounds: Lower: Upper:

Nonlinear constraint function: @unitdisk

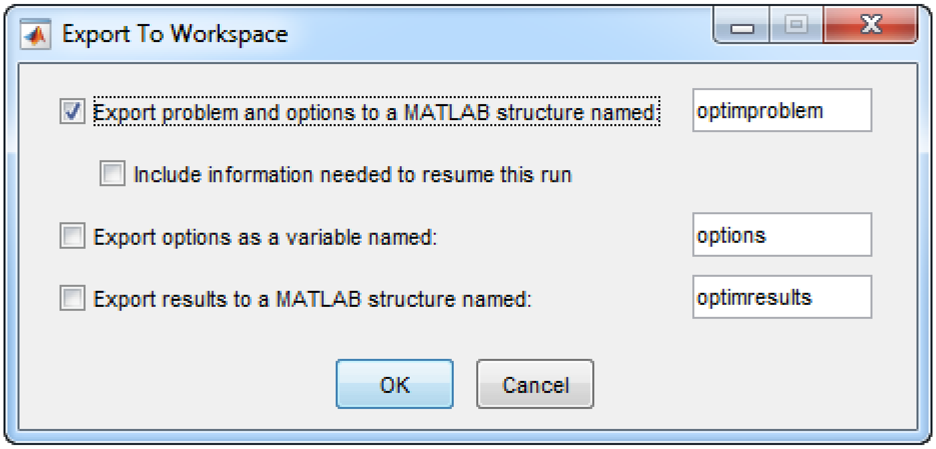
Derivatives: Approximated by solver

9 Set bounds, linear constraints, or local **Options**. For details on constraints, see “Writing Constraints”.

10 *Best practice: run the problem to verify the setup.*



11 Choose **File > Export to Workspace** and select **Export problem and options to a MATLAB structure named**


Export To Workspace

[✓] Export problem and options to a MATLAB structure named: optimproblem

[ ] Include information needed to resume this run

[ ] Export options as a variable named: options

[ ] Export results to a MATLAB structure named: optimresults

Example: Creating a Problem Structure with the Optimization App

This example minimizes the function from "Run the Solver" on page 3-20, subject to the constraint $x_1 + 2x_2 \geq 4$. The objective is
sixmin = $4x^2 - 2.1x^4 + x^6/3 + xy - 4y^2 + 4y^4$.

Use the `interior-point` algorithm of `fmincon`, and set the start point to `[2;3]`.

1 Write a function handle for the objective function.

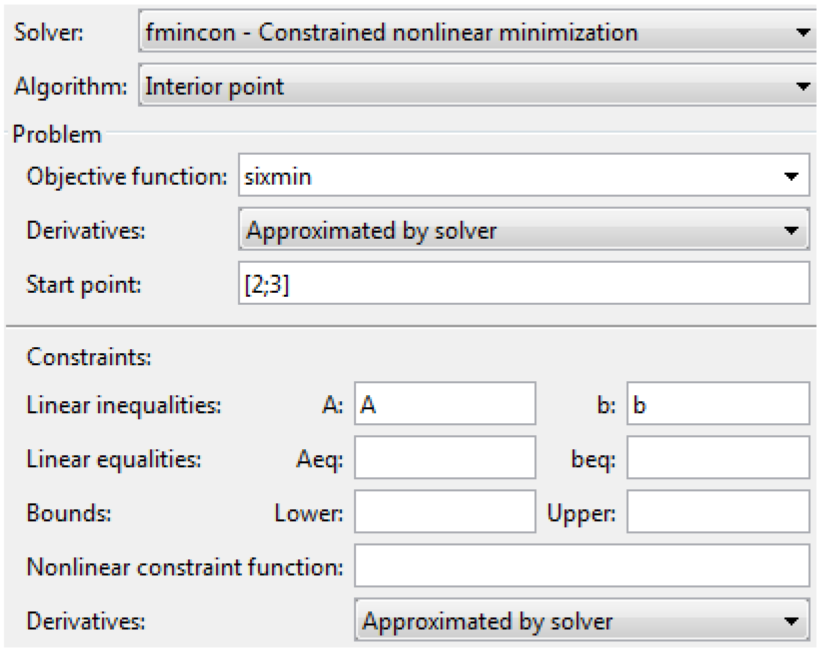
```
sixmin = @(x)(4*x(1)^2 - 2.1*x(1)^4 + x(1)^6/3 ...
    + x(1)*x(2) - 4*x(2)^2 + 4*x(2)^4);
```

2 Write the linear constraint matrices. Change the constraint to "less than" form:

```
A = [-1,-2];
b = -4;
```

3 Launch the Optimization app by entering `optimtool` at the MATLAB command line.

4 Set the solver, algorithm, objective, start point, and constraints.



Solver: fmincon - Constrained nonlinear minimization

Algorithm: Interior point

Problem

Objective function: sixmin

Derivatives: Approximated by solver

Start point: [2;3]

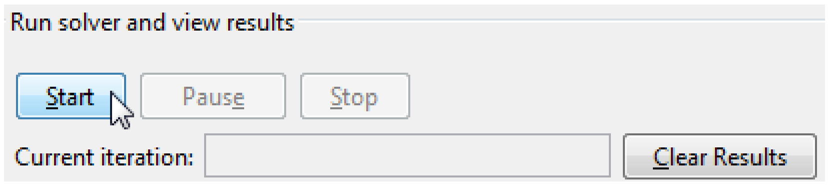
Constraints:

Linear inequalities:	A:	A	b:	b
Linear equalities:	Aeq:		beq:	
Bounds:	Lower:		Upper:	

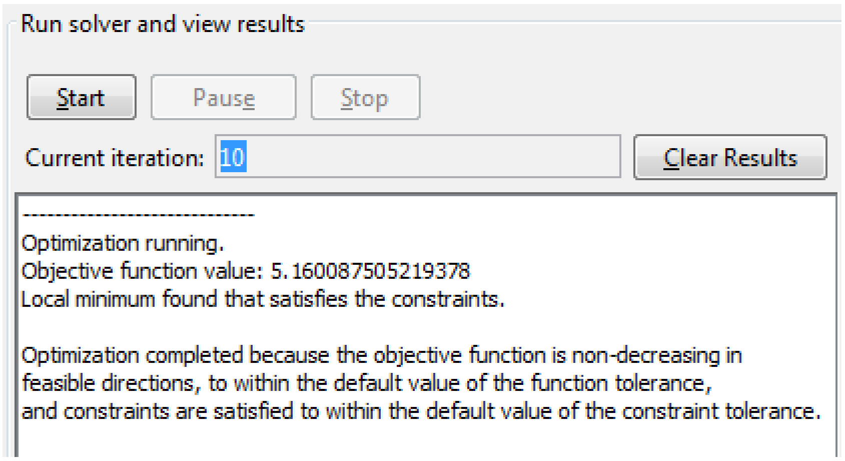
Nonlinear constraint function:

Derivatives: Approximated by solver

5 *Best practice: run the problem to verify the setup.*

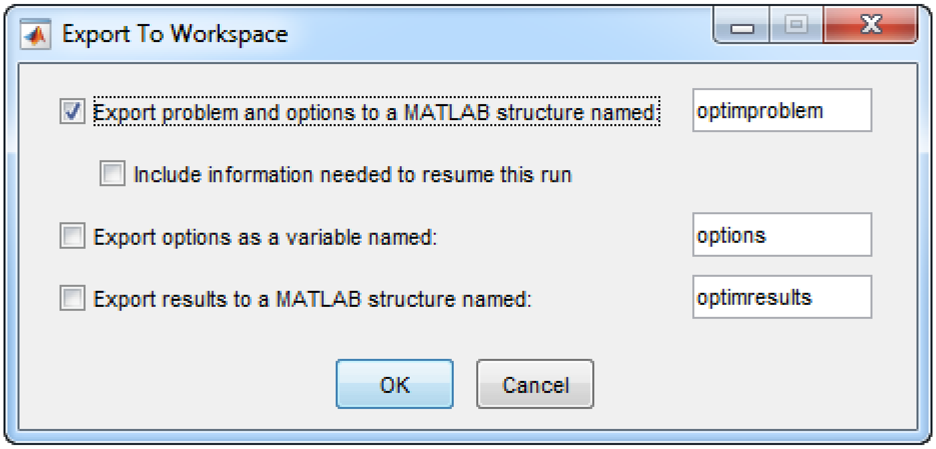


The problem runs successfully.



```
------------------------------
Optimization running.
Objective function value: 5.160087505219378
Local minimum found that satisfies the constraints.

Optimization completed because the objective function is non-decreasing in
feasible directions, to within the default value of the function tolerance,
and constraints are satisfied to within the default value of the constraint tolerance.
```

6 Choose **File > Export to Workspace** and select **Export problem and options to a MATLAB structure named**



[✓] Export problem and options to a MATLAB structure named: optimproblem

[ ] Include information needed to resume this run

[ ] Export options as a variable named: options

[ ] Export results to a MATLAB structure named: optimresults

Create Solver Object

In this section...
"What Is a Solver Object?" on page 3-14
"Properties (Global Options) of Solver Objects" on page 3-14
"Creating a Nondefault GlobalSearch Object" on page 3-16
"Creating a Nondefault MultiStart Object" on page 3-16

What Is a Solver Object?

A solver object contains your preferences for the global portion of the optimization.

You do not need to set any preferences. Create a `GlobalSearch` object named `gs` with default settings as follows:

```
gs = GlobalSearch;
```

Similarly, create a `MultiStart` object named `ms` with default settings as follows:

```
ms = MultiStart;
```

Properties (Global Options) of Solver Objects

Global options are properties of a `GlobalSearch` or `MultiStart` object.

Properties for both GlobalSearch and MultiStart

Property Name	Meaning
`Display`	Detail level of iterative display. Set to `'off'` for no display, `'final'` (default) for a report at the end of the run, or `'iter'` for reports as the solver progresses. For more information and examples, see "Iterative Display" on page 3-30.
`TolFun`	Solvers consider objective function values within `TolFun` of each other to be identical (not distinct). Default: `1e-6`. Solvers group solutions when the solutions satisfy both `TolFun` and `TolX` tolerances.
`TolX`	Solvers consider solutions within `TolX` distance of each other to be identical (not distinct). Default: `1e-6`. Solvers group

Property Name	Meaning
	solutions when the solutions satisfy both `TolFun` and `TolX` tolerances.
`MaxTime`	Solvers halt if the run exceeds `MaxTime` seconds, as measured by a clock (not processor seconds). Default: `Inf`
`StartPointsToRun`	Choose whether to run `'all'` (default) start points, only those points that satisfy `'bounds'`, or only those points that are feasible with respect to bounds and inequality constraints with `'bounds-ineqs'`. For an example, see "Optimize Using Only Feasible Start Points" on page 3-80.
`OutputFcns`	Functions to run after each local solver run. See "Output Functions for GlobalSearch and MultiStart" on page 3-37. Default: `[]`
`PlotFcns`	Plot functions to run after each local solver run. See "Plot Functions for GlobalSearch and MultiStart" on page 3-40. Default: `[]`

Properties for GlobalSearch

Property Name	Meaning
`NumTrialPoints`	Number of trial points to examine. Default: `1000`
`BasinRadiusFactor`	See "Properties" on page 11-38 for detailed descriptions of these properties.
`DistanceThresholdFactor`	
`MaxWaitCycle`	
`NumStageOnePoints`	
`PenaltyThresholdFactor`	

Properties for MultiStart

Property Name	Meaning
`UseParallel`	When `true`, `MultiStart` attempts to distribute start points to multiple processors for the local solver. Disable by setting to `false` (default). For details, see "How to Use Parallel Processing" on page 9-12. For an example, see "Parallel MultiStart" on page 3-89.

Creating a Nondefault GlobalSearch Object

Suppose you want to solve a problem and:

- Consider local solutions identical if they are within 0.01 of each other and the function values are within the default `TolFun` tolerance.
- Spend no more than 2000 seconds on the computation.

To solve the problem, create a `GlobalSearch` object `gs` as follows:

```
gs = GlobalSearch('TolX',0.01,'MaxTime',2000);
```

Creating a Nondefault MultiStart Object

Suppose you want to solve a problem such that:

- You consider local solutions identical if they are within 0.01 of each other and the function values are within the default `TolFun` tolerance.
- You spend no more than 2000 seconds on the computation.

To solve the problem, create a `MultiStart` object `ms` as follows:

```
ms = MultiStart('TolX',0.01,'MaxTime',2000);
```

Set Start Points for MultiStart

In this section...

Four Ways to Set Start Points

There are four ways you tell `MultiStart` which start points to use for the local solver:

- Pass a positive integer `k`. `MultiStart` generates `k - 1` start points as if using a `RandomStartPointSet` object and the `problem` structure. `MultiStart` also uses the `x0` start point from the `problem` structure, for a total of `k` start points.
- Pass a `RandomStartPointSet` object.
- Pass a `CustomStartPointSet` object.
- Pass a cell array of `RandomStartPointSet` and `CustomStartPointSet` objects. Pass a cell array if you have some specific points you want to run, but also want `MultiStart` to use other random start points.

Note: You can control whether `MultiStart` uses all start points, or only those points that satisfy bounds or other inequality constraints. For more information, see "Filter Start Points (Optional)" on page 3-51.

Positive Integer for Start Points

The syntax for running `MultiStart` for `k` start points is

```
[xmin,fmin,flag,outpt,allmins] = run(ms,problem,k);
```

The positive integer `k` specifies the number of start points `MultiStart` uses. `MultiStart` generates random start points using the dimension of the problem and bounds from the `problem` structure. `MultiStart` generates `k - 1` random start points, and also uses the `x0` start point from the `problem` structure.

RandomStartPointSet Object for Start Points

Create a `RandomStartPointSet` object as follows:

```
stpoints = RandomStartPointSet;
```

By default a `RandomStartPointSet` object generates 10 start points. Control the number of start points with the `NumStartPoints` property. For example, to generate 40 start points:

```
stpoints = RandomStartPointSet('NumStartPoints',40);
```

You can set an `ArtificialBound` for a `RandomStartPointSet`. This `ArtificialBound` works in conjunction with the bounds from the problem structure:

- If a component has no bounds, `RandomStartPointSet` uses a lower bound of `-ArtificialBound`, and an upper bound of `ArtificialBound`.
- If a component has a lower bound `lb` but no upper bound, `RandomStartPointSet` uses an upper bound of `lb + 2*ArtificialBound`.
- Similarly, if a component has an upper bound `ub` but no lower bound, `RandomStartPointSet` uses a lower bound of `ub - 2*ArtificialBound`.

For example, to generate `100` start points with an `ArtificialBound` of `50`:

```
stpoints = RandomStartPointSet('NumStartPoints',100, ...
    'ArtificialBound',50);
```

A `RandomStartPointSet` object generates start points with the same dimension as the `x0` point in the problem structure; see `list`.

CustomStartPointSet Object for Start Points

To use a specific set of starting points, package them in a `CustomStartPointSet` as follows:

1. Place the starting points in a matrix. Each row of the matrix represents one starting point. `MultiStart` runs all the rows of the matrix, subject to filtering with the `StartPointsToRun` property. For more information, see “MultiStart Algorithm” on page 3-50.
2. Create a `CustomStartPointSet` object from the matrix:

   ```
   tpoints = CustomStartPointSet(ptmatrix);
   ```

For example, create a set of 40 five-dimensional points, with each component of a point equal to 10 plus an exponentially distributed variable with mean 25:

```
pts = -25*log(rand(40,5)) + 10;
tpoints = CustomStartPointSet(pts);
```

To get the original matrix of points from a `CustomStartPointSet` object, use the `list` method:

```
pts = list(tpoints); % Assumes tpoints is a CustomStartPointSet
```

A `CustomStartPointSet` has two properties: `DimStartPoints` and `NumStartPoints`. You can use these properties to query a `CustomStartPointSet` object. For example, the `tpoints` object in the example has the following properties:

```
tpoints.DimStartPoints
ans =
     5

tpoints.NumStartPoints
ans =
    40
```

Cell Array of Objects for Start Points

To use a specific set of starting points along with some randomly generated points, pass a cell array of `RandomStartPointSet` or `CustomStartPointSet` objects.

For example, to use both the 40 specific five-dimensional points of "CustomStartPointSet Object for Start Points" on page 3-18 and 40 additional five-dimensional points from `RandomStartPointSet`:

```
pts = -25*log(rand(40,5)) + 10;
tpoints = CustomStartPointSet(pts);
rpts = RandomStartPointSet('NumStartPoints',40);
allpts = {tpoints,rpts};
```

Run `MultiStart` with the `allpts` cell array:

```
% Assume ms and problem exist
[xmin,fmin,flag,outpt,allmins] = run(ms,problem,allpts);
```

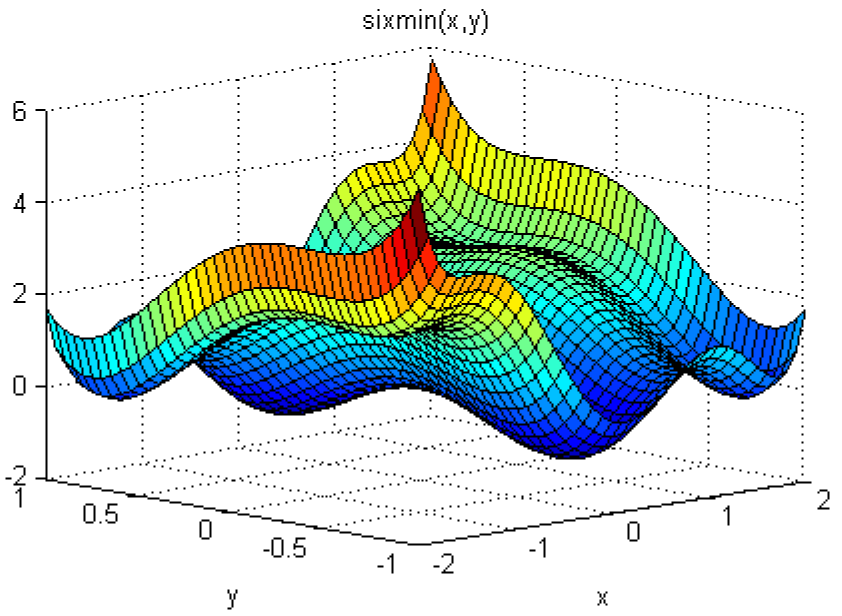
Run the Solver

In this section...

Optimize by Calling run

Running a solver is nearly identical for `GlobalSearch` and `MultiStart`. The only difference in syntax is `MultiStart` takes an additional input describing the start points.

For example, suppose you want to find several local minima of the `sixmin` function sixmin $= 4x^2 - 2.1x^4 + x^6/3 + xy - 4y^2 + 4y^4$.



This function is also called the six-hump camel back function [3]. All the local minima lie in the region $-3 \le x,y \le 3$.

Example of Run with GlobalSearch

To find several local minima of the `sixmin` function using `GlobalSearch`, enter:

```
% % Set the random stream to get exactly the same output
% rng(14,'twister')
gs = GlobalSearch;
opts = optimoptions(@fmincon,'Algorithm','interior-point');
sixmin = @(x)(4*x(1)^2 - 2.1*x(1)^4 + x(1)^6/3 ...
    + x(1)*x(2) - 4*x(2)^2 + 4*x(2)^4);
problem = createOptimProblem('fmincon','x0',[-1,2],...
    'objective',sixmin,'lb',[-3,-3],'ub',[3,3],...
    'options',opts);
[xming,fming,flagg,outptg,manyminsg] = run(gs,problem);
```

The output of the run (which varies, based on the random seed):

```
xming,fming,flagg,outptg,manyminsg
xming =
    0.0898   -0.7127

fming =
   -1.0316

flagg =
     1

outptg =
                funcCount: 2131
         localSolverTotal: 3
       localSolverSuccess: 3
    localSolverIncomplete: 0
    localSolverNoSolution: 0
                  message: 'GlobalSearch stopped because it analyzed all the trial po

manyminsg =
  1x2 GlobalOptimSolution array with properties:

    X
    Fval
    Exitflag
    Output
    X0
```

Example of Run with MultiStart

To find several local minima of the `sixmin` function using 50 runs of `fmincon` with `MultiStart`, enter:

```
% % Set the random stream to get exactly the same output
% rng(14,'twister')
ms = MultiStart;
opts = optimoptions(@fmincon,'Algorithm','interior-point');
sixmin = @(x)(4*x(1)^2 - 2.1*x(1)^4 + x(1)^6/3 ...
    + x(1)*x(2) - 4*x(2)^2 + 4*x(2)^4);
problem = createOptimProblem('fmincon','x0',[-1,2],...
    'objective',sixmin,'lb',[-3,-3],'ub',[3,3],...
    'options',opts);
[xminm,fminm,flagm,outptm,manyminsm] = run(ms,problem,50);
```

The output of the run (which varies based on the random seed):

```
xminm,fminm,flagm,outptm,manyminsm

xminm =
    0.0898    -0.7127

fminm =
   -1.0316

flagm =
     1

outptm =
                funcCount: 2034
         localSolverTotal: 50
       localSolverSuccess: 50
    localSolverIncomplete: 0
    localSolverNoSolution: 0
                  message: 'MultiStart completed the runs from all start points.

All...'

manyminsm =
  1x6 GlobalOptimSolution array with properties:

    X
    Fval
```

```
Exitflag
Output
X0
```

In this case, `MultiStart` located all six local minima, while `GlobalSearch` located two. For pictures of the `MultiStart` solutions, see "Visualize the Basins of Attraction" on page 3-34.

Single Solution

You obtain the single best solution found during the run by calling `run` with the syntax

```
[x fval eflag output] = run(...);
```

- `x` is the location of the local minimum with smallest objective function value.
- `fval` is the objective function value evaluated at `x`.
- `eflag` is an exit flag for the global solver. Values:

  Global Solver Exit Flags

`2`	At least one local minimum found. Some runs of the local solver converged (had positive exit flag).
`1`	At least one local minimum found. All runs of the local solver converged (had positive exit flag).
`0`	No local minimum found. Local solver called at least once, and at least one local solver exceeded the `MaxIter` or `MaxFunEvals` tolerances.
`-1`	Solver stopped by output function or plot function.
`-2`	No feasible local minimum found.
`-5`	`MaxTime` limit exceeded.
`-8`	No solution found. All runs had local solver exit flag `-1` or smaller.
`-10`	Failures encountered in user-provided functions.

- `output` is a structure with details about the multiple runs of the local solver. For more information, see "Global Output Structures" on page 3-33.

The list of outputs is for the case `eflag > 0`. If `eflag <= 0`, then `x` is the following:

- If some local solutions are feasible, `x` represents the location of the lowest objective function value. "Feasible" means the constraint violations are smaller than `problem.options.TolCon`.
- If no solutions are feasible, `x` is the solution with lowest infeasibility.
- If no solutions exist, `x`, `fval`, and `output` are empty (`[]`).

Multiple Solutions

In this section...

About Multiple Solutions

You obtain multiple solutions in an object by calling `run` with the syntax

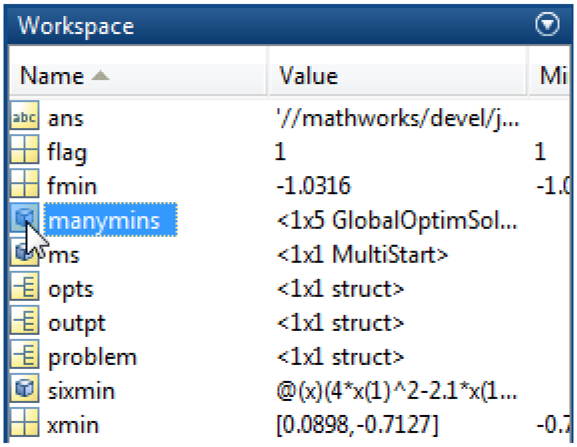
```
[x fval eflag output manymins] = run(...);
```

`manymins` is a vector of solution objects; see `GlobalOptimSolution`. The `manymins` vector is in order of objective function value, from lowest (best) to highest (worst). Each solution object contains the following properties (fields):

- `X` — a local minimum
- `Fval` — the value of the objective function at `X`
- `Exitflag` — the exit flag for the local solver (described in the local solver function reference page: `fmincon`, `fminunc`, `lsqcurvefit`, or `lsqnonlin`)
- `Output` — an output structure for the local solver (described in the local solver function reference page: `fmincon`, `fminunc`, `lsqcurvefit`, or `lsqnonlin`)
- `X0` — a cell array of start points that led to the solution point `X`

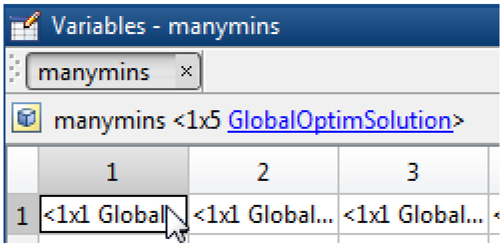
There are several ways to examine the vector of solution objects:

- In the MATLAB Workspace Browser. Double-click the solution object, and then double-click the resulting display in the Variables editor.



Workspace

Name	Value	Mi
ans	'//mathworks/devel/j...	
flag	1	1
fmin	-1.0316	-1.(
manymins	<1x5 GlobalOptimSol...	
ms	<1x1 MultiStart>	
opts	<1x1 struct>	
outpt	<1x1 struct>	
problem	<1x1 struct>	
sixmin	@(x)(4*x(1)^2-2.1*x(1...	
xmin	[0.0898,-0.7127]	-0.7

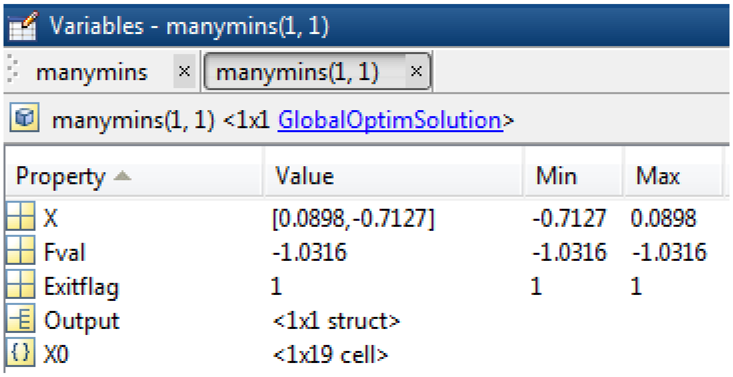


Variables - manymins

manymins ×

manymins <1x5 GlobalOptimSolution>

	1	2	3
1	<1x1 Global	<1x1 Global...	<1x1 Global...



Variables - manymins(1, 1)

manymins × manymins(1, 1) ×

manymins(1, 1) <1x1 GlobalOptimSolution>

Property	Value	Min	Max
X	[0.0898,-0.7127]	-0.7127	0.0898
Fval	-1.0316	-1.0316	-1.0316
Exitflag	1	1	1
Output	<1x1 struct>		
X0	<1x19 cell>		

- Using dot notation. `GlobalOptimSolution` properties are capitalized. Use proper capitalization to access the properties.

For example, to find the vector of function values, enter:

```
fcnvals = [manymins.Fval]

fcnvals =
   -1.0316   -0.2155         0
```

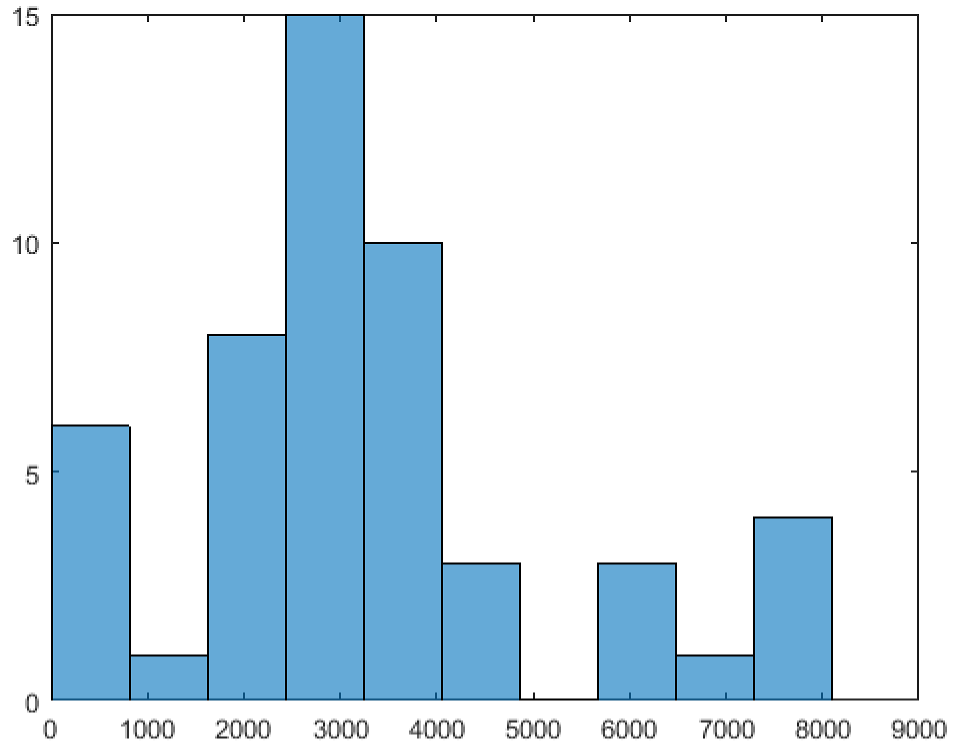
To get a cell array of all the start points that led to the lowest function value (the first element of `manymins`), enter:

```
smallX0 = manymins(1).X0
```

- Plot some field values. For example, to see the range of resulting `Fval`, enter:

  ```
  histogram([manymins.Fval],10)
  ```

  This results in a histogram of the computed function values. (The figure shows a histogram from a different example than the previous few figures.)



Change the Definition of Distinct Solutions

You might find out, after obtaining multiple local solutions, that your tolerances were not appropriate. You can have many more local solutions than you want, spaced too closely together. Or you can have fewer solutions than you want, with `GlobalSearch` or `MultiStart` clumping together too many solutions.

To deal with this situation, run the solver again with different tolerances. The `TolX` and `TolFun` tolerances determine how the solvers group their outputs into the `GlobalOptimSolution` vector. These tolerances are properties of the `GlobalSearch` or `MultiStart` object.

For example, suppose you want to use the `active-set` algorithm in `fmincon` to solve the problem in “Example of Run with MultiStart” on page 3-22. Further suppose that you want to have tolerances of `0.01` for both `TolX` and `TolFun`. The `run` method groups local solutions whose objective function values are within `TolFun` of each other, and which are also less than `TolX` apart from each other. To obtain the solution:

```
% % Set the random stream to get exactly the same output
% rng(14,'twister')
ms = MultiStart('TolFun',0.01,'TolX',0.01);
opts = optimoptions(@fmincon,'Algorithm','active-set');
sixmin = @(x)(4*x(1)^2 - 2.1*x(1)^4 + x(1)^6/3 ...
    + x(1)*x(2) - 4*x(2)^2 + 4*x(2)^4);
problem = createOptimProblem('fmincon','x0',[-1,2],...
    'objective',sixmin,'lb',[-3,-3],'ub',[3,3],...
    'options',opts);
[xminm,fminm,flagm,outptm,someminsm] = run(ms,problem,50);

MultiStart completed the runs from all start points.

All 50 local solver runs converged with a
positive local solver exit flag.

someminsm

someminsm =

  1x5 GlobalOptimSolution

  Properties:
    X
    Fval
    Exitflag
    Output
    X0
```

In this case, `MultiStart` generated five distinct solutions. Here “distinct” means that the solutions are more than 0.01 apart in either objective function value or location.

Iterative Display

In this section...

Types of Iterative Display

Iterative display gives you information about the progress of solvers during their runs.

There are two types of iterative display:

- Global solver display
- Local solver display

Both types appear at the command line, depending on global and local options.

Obtain local solver iterative display by setting the `Display` option in the `problem.options` structure to `'iter'` or `'iter-detailed'` with `optimoptions`. For more information, see "Iterative Display" in the Optimization Toolbox documentation.

Obtain global solver iterative display by setting the `Display` property in the `GlobalSearch` or `MultiStart` object to `'iter'`.

Global solvers set the default `Display` option of the local solver to `'off'`, unless the problem structure has a value for this option. Global solvers do not override any setting you make for local options.

Note: Setting the local solver `Display` option to anything other than `'off'` can produce a great deal of output. The default `Display` option created by `optimoptions(@`*`solver`*`)` is `'final'`.

Examine Types of Iterative Display

Run the example described in "Run the Solver" on page 3-20 using `GlobalSearch` with `GlobalSearch` iterative display:

```
% % Set the random stream to get exactly the same output
% rng(14,'twister')
gs = GlobalSearch('Display','iter');
opts = optimoptions(@fmincon,'Algorithm','interior-point');
sixmin = @(x)(4*x(1)^2 - 2.1*x(1)^4 + x(1)^6/3 ...
    + x(1)*x(2) - 4*x(2)^2 + 4*x(2)^4);
problem = createOptimProblem('fmincon','x0',[-1,2],...
    'objective',sixmin,'lb',[-3,-3],'ub',[3,3],...
    'options',opts);
[xming,fming,flagg,outptg,manyminsg] = run(gs,problem);

 Num Pts                 Best       Current    Threshold        Local        Local
Analyzed  F-count        f(x)       Penalty      Penalty         f(x)     exitflag        Procedure
       0       34      -1.032                                  -1.032            1    Initial Point
     200     1291      -1.032                                 -0.2155            1    Stage 1 Local
     300     1393      -1.032         248.7      -0.2137                              Stage 2 Search
     400     1493      -1.032           278        1.134                              Stage 2 Search
     446     1577      -1.032           1.6        2.073      -0.2155            1    Stage 2 Local
     500     1631      -1.032         9.055       0.3214                              Stage 2 Search
     600     1731      -1.032       -0.7299      -0.7686                              Stage 2 Search
     700     1831      -1.032        0.3191      -0.7431                              Stage 2 Search
     800     1931      -1.032         296.4       0.4577                              Stage 2 Search
     900     2031      -1.032         10.68       0.5116                              Stage 2 Search
    1000     2131      -1.032       -0.9207      -0.9254                              Stage 2 Search

GlobalSearch stopped because it analyzed all the trial points.

All 3 local solver runs converged with a positive local solver exit flag.
```

Run the same example without `GlobalSearch` iterative display, but with `fmincon` iterative display:

```
gs.Display = 'final';
problem.options.Display = 'iter';
[xming,fming,flagg,outptg,manyminsg] = run(gs,problem);

                                            First-order      Norm of
 Iter F-count            f(x)  Feasibility   optimality         step
    0       3    4.823333e+001   0.000e+000   1.088e+002
    1       7    2.020476e+000   0.000e+000   2.176e+000   2.488e+000
    2      10    6.525252e-001   0.000e+000   1.937e+000   1.886e+000
    3      13   -8.776121e-001   0.000e+000   9.076e-001   8.539e-001
    4      16   -9.121907e-001   0.000e+000   9.076e-001   1.655e-001
    5      19   -1.009367e+000   0.000e+000   7.326e-001   8.558e-002
    6      22   -1.030423e+000   0.000e+000   2.172e-001   6.670e-002
    7      25   -1.031578e+000   0.000e+000   4.278e-002   1.444e-002
    8      28   -1.031628e+000   0.000e+000   8.777e-003   2.306e-003
    9      31   -1.031628e+000   0.000e+000   8.845e-005   2.750e-004
   10      34   -1.031628e+000   0.000e+000   8.744e-007   1.354e-006

Local minimum found that satisfies the constraints.

Optimization completed because the objective function is non-decreasing in
feasible directions, to within the selected value of the function tolerance,
and constraints were satisfied to within the selected value of the constraint tolerance.

<stopping criteria details>
                                            First-order      Norm of
 Iter F-count            f(x)  Feasibility   optimality         step
    0       3    -1.980435e-02    0.000e+00    1.996e+00

... MANY ITERATIONS DELETED ...
```

```
    8      33   -1.031628e+00    0.000e+00    8.742e-07    2.287e-07

Local minimum found that satisfies the constraints.

Optimization completed because the objective function is non-decreasing in
feasible directions, to within the selected value of the function tolerance,
and constraints were satisfied to within the selected value of the constraint tolerance.

<stopping criteria details>

GlobalSearch stopped because it analyzed all the trial points.

All 4 local solver runs converged with a positive local solver exit flag.
```

Setting `GlobalSearch` iterative display, as well as `fmincon` iterative display, yields both displays intermingled.

For an example of iterative display in a parallel environment, see "Parallel MultiStart" on page 3-89.

Global Output Structures

`run` can produce two types of output structures:

- A global output structure. This structure contains information about the overall run from multiple starting points. Details follow.
- Local solver output structures. The vector of `GlobalOptimSolution` objects contains one such structure in each element of the vector. For a description of this structure, see "Output Structures" in the Optimization Toolbox documentation, or the function reference pages for the local solvers: `fmincon`, `fminunc`, `lsqcurvefit`, or `lsqnonlin`.

Global Output Structure

Field	Meaning
`funcCount`	Total number of calls to user-supplied functions (objective or nonlinear constraint)
`localSolverTotal`	Number of local solver runs started
`localSolverSuccess`	Number of local solver runs that finished with a positive exit flag
`localSolverIncomplete`	Number of local solver runs that finished with a `0` exit flag
`localSolverNoSolution`	Number of local solver runs that finished with a negative exit flag
`message`	`GlobalSearch` or `MultiStart` exit message

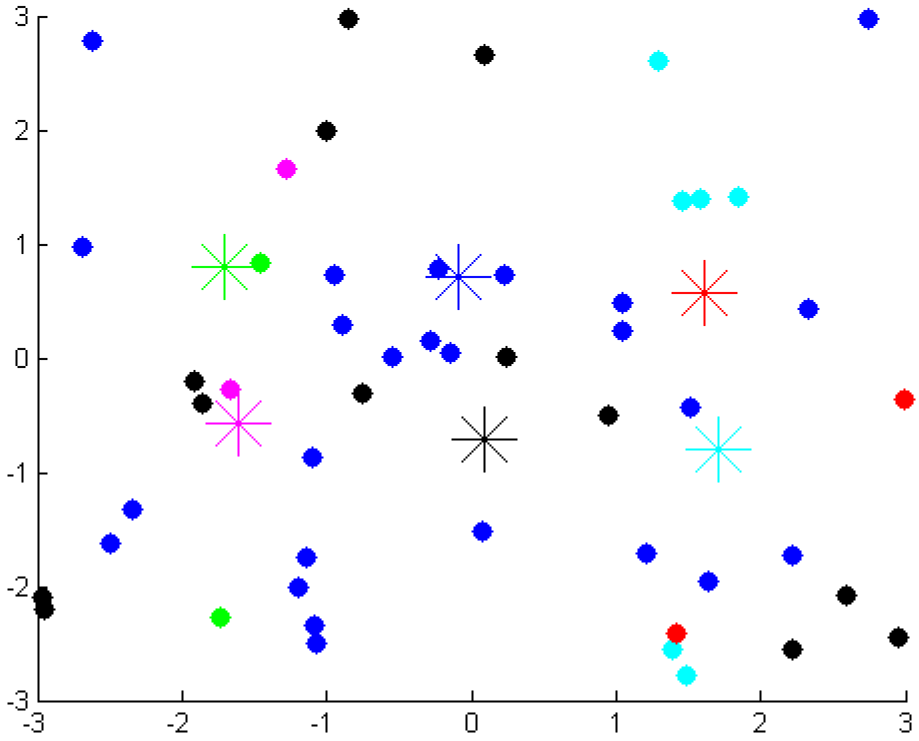
A positive exit flag from a local solver generally indicates a successful run. A negative exit flag indicates a failure. A `0` exit flag indicates that the solver stopped by exceeding the iteration or function evaluation limit. For more information, see "Exit Flags and Exit Messages" or "Tolerances and Stopping Criteria" in the Optimization Toolbox documentation.

Visualize the Basins of Attraction

Which start points lead to which basin? For a steepest descent solver, nearby points generally lead to the same basin; see “Basins of Attraction” on page 1-13. However, for Optimization Toolbox solvers, basins are more complicated.

Plot the `MultiStart` start points from the example, “Example of Run with MultiStart” on page 3-22, color-coded with the basin where they end.

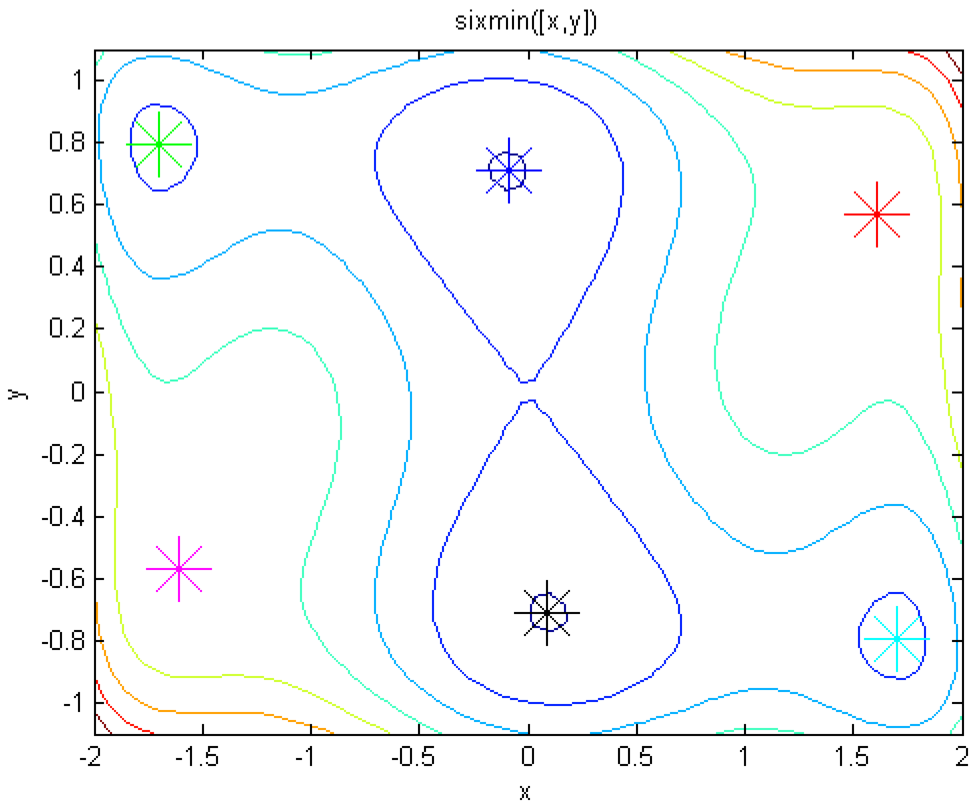
```
% rng(14,'twister')
% Uncomment the previous line to get the same output
ms = MultiStart;
opts = optimoptions(@fmincon,'Algorithm','interior-point');
sixmin = @(x)(4*x(1)^2 - 2.1*x(1)^4 + x(1)^6/3 ...
+ x(1)*x(2) - 4*x(2)^2 + 4*x(2)^4);
problem = createOptimProblem('fmincon','x0',[-1,2],...
'objective',sixmin,'lb',[-3,-3],'ub',[3,3],...
'options',opts);
[xminm,fminm,flagm,outptm,manyminsm] = run(ms,problem,50);

possColors = 'kbgcrm';
hold on
for i = 1:size(manyminsm,2)

    % Color of this line
    cIdx = rem(i-1, length(possColors)) + 1;
    color = possColors(cIdx);

    % Plot start points
    u = manyminsm(i).X0;
    x0ThisMin = reshape([u{:}], 2, length(u));
    plot(x0ThisMin(1, :), x0ThisMin(2, :), '.', ...
        'Color',color,'MarkerSize',25);

    % Plot the basin with color i
    plot(manyminsm(i).X(1), manyminsm(i).X(2), '*', ...
        'Color', color, 'MarkerSize',25);
end % basin center marked with a *, start points with dots
hold off
```



The figure shows the centers of the basins by colored * symbols. Start points with the same color as the * symbol converge to the center of the * symbol.

Start points do not always converge to the closest basin. For example, the red points are closer to the cyan basin center than to the red basin center. Also, many black and blue start points are closer to the opposite basin centers.

The magenta and red basins are shallow, as you can see in the following contour plot.


sixmin([x,y])
1
0.8
0.6
0.4
0.2
0
-0.2
-0.4
-0.6
-0.8
-1
y
-2
-1.5
-1
-0.5
0
0.5
1
1.5
2
x

Output Functions for GlobalSearch and MultiStart

In this section...

What Are Output Functions?

Output functions allow you to examine intermediate results in an optimization. Additionally, they allow you to halt a solver programmatically.

There are two types of output functions, like the two types of output structures:

- Global output functions run after each local solver run. They also run when the global solver starts and ends.
- Local output functions run after each iteration of a local solver. See “Output Functions” in the Optimization Toolbox documentation.

To use global output functions:

- Write output functions using the syntax described in “OutputFcns” on page 10-3.
- Set the `OutputFcns` property of your `GlobalSearch` or `MultiStart` solver to the function handle of your output function. You can use multiple output functions by setting the `OutputFcns` property to a cell array of function handles.

GlobalSearch Output Function

This output function stops `GlobalSearch` after it finds five distinct local minima with positive exit flags, or after it finds a local minimum value less than `0.5`. The output function uses a persistent local variable, `foundLocal`, to store the local results. `foundLocal` enables the output function to determine whether a local solution is distinct from others, to within a tolerance of `1e-4`.

To store local results using nested functions instead of persistent variables, see “Example of a Nested Output Function” in the MATLAB Mathematics documentation.

1. Write the output function using the syntax described in “OutputFcns” on page 10-3.

```
function stop = StopAfterFive(optimValues, state)
persistent foundLocal
stop = false;
switch state
    case 'init'
        foundLocal = []; % initialized as empty
    case 'iter'
        newf = optimValues.localsolution.Fval;
        eflag = optimValues.localsolution.Exitflag;
        % Now check if the exit flag is positive and
        % the new value differs from all others by at least 1e-4
        % If so, add the new value to the newf list
        if eflag > 0 && all(abs(newf - foundLocal) > 1e-4)
            foundLocal = [foundLocal;newf];
            % Now check if the latest value added to foundLocal
            % is less than 1/2
            % Also check if there are 5 local minima in foundLocal
            % If so, then stop
            if foundLocal(end) < 0.5 || length(foundLocal) >= 5
                stop = true;
            end
        end
end
```

2. Save `StopAfterFive.m` as a file in a folder on your MATLAB path.
3. Write the objective function and create an optimization problem structure as in "Find Global or Multiple Local Minima" on page 3-72.

```
function f = sawtoothxy(x,y)
[t r] = cart2pol(x,y); % change to polar coordinates
h = cos(2*t - 1/2)/2 + cos(t) + 2;
g = (sin(r) - sin(2*r)/2 + sin(3*r)/3 - sin(4*r)/4 + 4) ...
    .*r.^2./(r+1);
f = g.*h;
end
```

4. Save `sawtoothxy.m` as a file in a folder on your MATLAB path.
5. At the command line, create the problem structure:

```
problem = createOptimProblem('fmincon',...
    'objective',@(x)sawtoothxy(x(1),x(2)),...
    'x0',[100,-50],'options',...
    optimoptions(@fmincon,'Algorithm','sqp'));
```

6. Create a `GlobalSearch` object with `@StopAfterFive` as the output function, and set the iterative display property to `'iter'`.

```
gs = GlobalSearch('OutputFcns',@StopAfterFive,'Display','iter');
```

7. (Optional) To get the same answer as this example, set the default random number stream.

```
rng default
```

8. Run the problem.

```
[x fval] = run(gs,problem)

 Num Pts                 Best       Current    Threshold        Local        Local                 
Analyzed  F-count        f(x)       Penalty      Penalty         f(x)     exitflag        Procedure
       0      200       555.7                                   555.7            0    Initial Point
     200     1479   1.547e-15                               1.547e-15            1    Stage 1 Local

GlobalSearch stopped by the output or plot function.

1 out of 2 local solver runs converged with a positive local solver exit flag.

x =

   1.0e-07 *

    0.0414    0.1298


fval =

   1.5467e-15
```

The run stopped early because `GlobalSearch` found a point with a function value less than `0.5`.

No Parallel Output Functions

While `MultiStart` can run in parallel, it does not support global output functions and plot functions in parallel. Furthermore, while local output functions and plot functions run on workers when `MultiStart` runs in parallel, the effect differs from running serially. Local output and plot functions do not create a display when running on workers. You do not see any other effects of output and plot functions until the worker passes its results to the client (the originator of the `MultiStart` parallel jobs).

For information on running `MultiStart` in parallel, see “Parallel Computing”.

Plot Functions for GlobalSearch and MultiStart

In this section...

What Are Plot Functions?

The `PlotFcns` field of the `options` structure specifies one or more functions that an optimization function calls at each iteration. Plot functions plot various measures of progress while the algorithm executes. Pass a function handle or cell array of function handles. The structure of a plot function is the same as the structure of an output function. For more information on this structure, see "OutputFcns" on page 10-3.

Plot functions are specialized output functions (see "Output Functions for GlobalSearch and MultiStart" on page 3-37). There are two predefined plot functions:

- `@gsplotbestf` plots the best objective function value.
- `@gsplotfunccount` plots the number of function evaluations.

Plot function windows have **Pause** and **Stop** buttons. By default, all plots appear in one window.

To use global plot functions:

- Write plot functions using the syntax described in "OutputFcns" on page 10-3.
- Set the `PlotFcns` property of your `GlobalSearch` or `MultiStart` object to the function handle of your plot function. You can use multiple plot functions by setting the `PlotFcns` property to a cell array of function handles.

Details of Built-In Plot Functions

The built-in plot functions have characteristics that can surprise you.

- `@gsplotbestf` can have plots that are not strictly decreasing. This is because early values can result from local solver runs with negative exit flags (such as infeasible solutions). A subsequent local solution with positive exit flag is better even if its function value is higher. Once a local solver returns a value with a positive exit flag, the plot is monotone decreasing.

- `@gsplotfunccount` might not plot the total number of function evaluations. This is because `GlobalSearch` can continue to perform function evaluations after it calls the plot function for the last time. For more information, see "GlobalSearch Algorithm" on page 3-46Properties for GlobalSearch.

MultiStart Plot Function

This example plots the number of local solver runs it takes to obtain a better local minimum for `MultiStart`. The example also uses a built-in plot function to show the current best function value.

The example problem is the same as in "Find Global or Multiple Local Minima" on page 3-72, with additional bounds.

The example uses persistent variables to store previous best values. The plot function examines the best function value after each local solver run, available in the `bestfval` field of the `optimValues` structure. If the value is not lower than the previous best, the plot function adds 1 to the number of consecutive calls with no improvement and draws a bar chart. If the value is lower than the previous best, the plot function starts a new bar in the chart with value 1. Before plotting, the plot function takes a logarithm of the number of consecutive calls. The logarithm helps keep the plot legible, since some values can be much larger than others.

To store local results using nested functions instead of persistent variables, see "Example of a Nested Output Function" in the MATLAB Mathematics documentation.

1 Write the objective function:

```
function f = sawtoothxy(x,y)
[t r] = cart2pol(x,y); % change to polar coordinates
h = cos(2*t - 1/2)/2 + cos(t) + 2;
g = (sin(r) - sin(2*r)/2 + sin(3*r)/3 - sin(4*r)/4 + 4) ...
    .*r.^2./(r+1);
f = g.*h;
```

2 Save `sawtoothxy.m` as a file in a folder on your MATLAB path.

3 Write the plot function:

```
function stop = NumberToNextBest(optimValues, state)

persistent bestfv bestcounter

stop = false;
switch state
```

```
        case 'init'
            % Initialize variable to record best function value.
            bestfv = [];

            % Initialize counter to record number of
            % local solver runs to find next best minimum.
            bestcounter = 1;

            % Create the histogram.
            bar(log(bestcounter),'tag','NumberToNextBest');
            xlabel('Number of New Best Fval Found');
            ylabel('Log Number of Local Solver Runs');
            title('Number of Local Solver Runs to Find Lower Minimum')
        case 'iter'
            % Find the axes containing the histogram.
            NumToNext = ...
              findobj(get(gca,'Children'),'Tag','NumberToNextBest');

            % Update the counter that records number of local
            % solver runs to find next best minimum.
            if ~isequal(optimValues.bestfval, bestfv)
                bestfv = optimValues.bestfval;
                bestcounter = [bestcounter 1];
            else
                bestcounter(end) = bestcounter(end) + 1;
            end

            % Update the histogram.
            set(NumToNext,'Ydata',log(bestcounter))
    end
```

4 Save `NumberToNextBest.m` as a file in a folder on your MATLAB path.

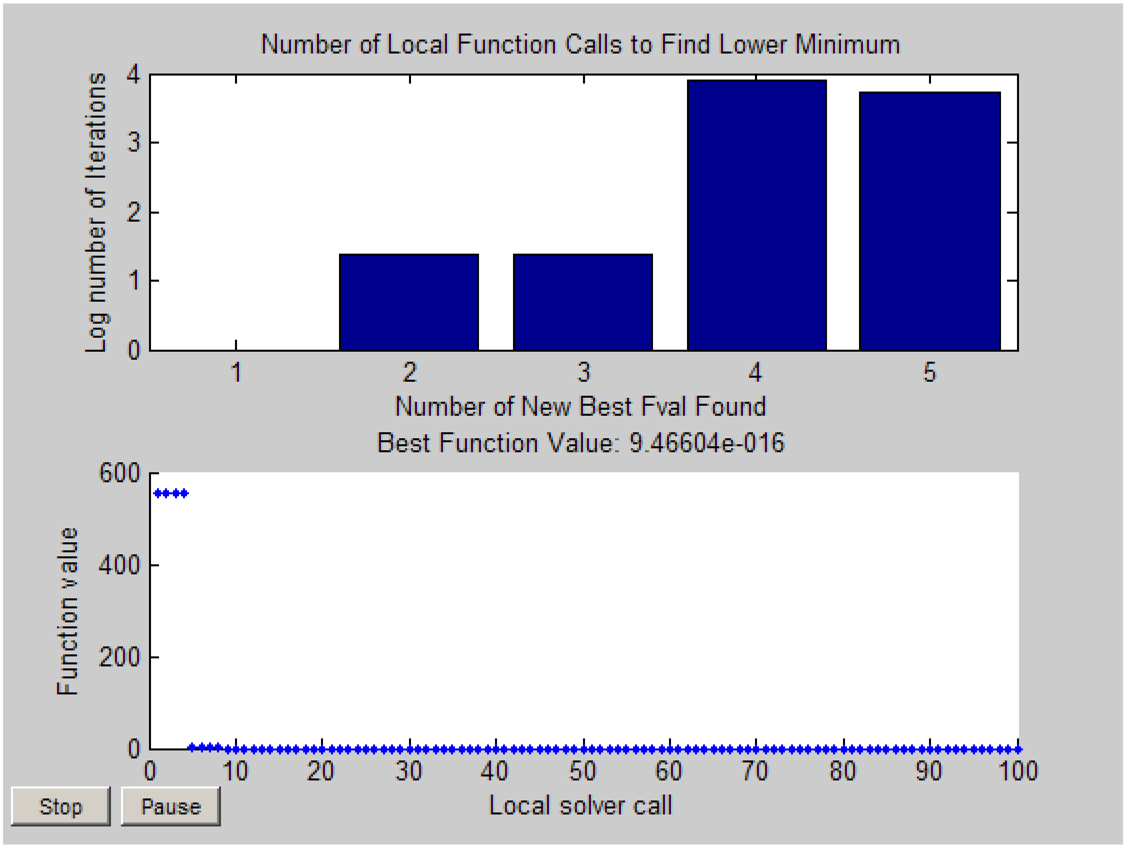
5 Create the problem structure and global solver. Set lower bounds of `[-3e3,-4e3]`, upper bounds of `[4e3,3e3]` and set the global solver to use the plot functions:

```
problem = createOptimProblem('fmincon',...
    'objective',@(x)sawtoothxy(x(1),x(2)),...
    'x0',[100,-50],'lb',[-3e3 -4e3],...
    'ub',[4e3,3e3],'options',...
    optimoptions(@fmincon,'Algorithm','sqp'));

ms = MultiStart('PlotFcns',{@NumberToNextBest,@gsplotbestf});
```

6 Run the global solver for 100 local solver runs:

```
[x fv] = run(ms,problem,100);
```

7 The plot functions produce the following figure (your results can differ, since the solution process is stochastic):



No Parallel Plot Functions

While `MultiStart` can run in parallel, it does not support global output functions and plot functions in parallel. Furthermore, while local output functions and plot functions run on workers when `MultiStart` runs in parallel, the effect differs from running serially. Local output and plot functions do not create a display when running on workers. You do not see any other effects of output and plot functions until the worker passes its results to the client (the originator of the `MultiStart` parallel jobs).

For information on running `MultiStart` in parallel, see “Parallel Computing”.

How GlobalSearch and MultiStart Work

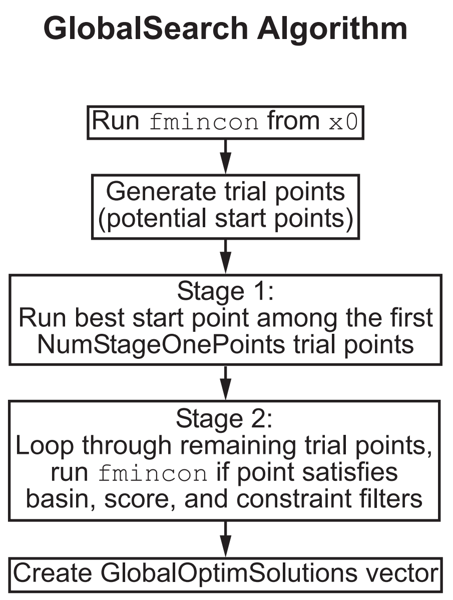
In this section...

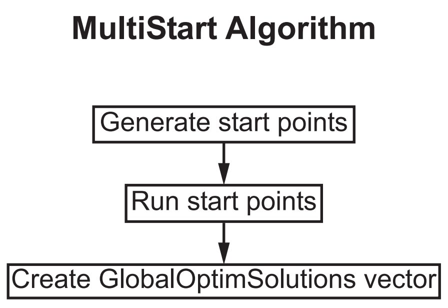
Multiple Runs of a Local Solver

`GlobalSearch` and `MultiStart` have similar approaches to finding global or multiple minima. Both algorithms start a local solver (such as `fmincon`) from multiple start points. The algorithms use multiple start points to sample multiple basins of attraction. For more information, see "Basins of Attraction" on page 1-13.

Differences Between the Solver Objects

GlobalSearch and MultiStart Algorithm Overview contains a sketch of the `GlobalSearch` and `MultiStart` algorithms.


GlobalSearch Algorithm
Run fmincon from x0
Generate trial points (potential start points)
Stage 1: Run best start point among the first NumStageOnePoints trial points
Stage 2: Loop through remaining trial points, run fmincon if point satisfies basin, score, and constraint filters
Create GlobalOptimSolutions vector



GlobalSearch and MultiStart Algorithm Overview

The main differences between `GlobalSearch` and `MultiStart` are:

- `GlobalSearch` uses a scatter-search mechanism for generating start points. `MultiStart` uses uniformly distributed start points within bounds, or user-supplied start points.
- `GlobalSearch` analyzes start points and rejects those points that are unlikely to improve the best local minimum found so far. `MultiStart` runs all start points (or, optionally, all start points that are feasible with respect to bounds or inequality constraints).
- `MultiStart` gives a choice of local solver: `fmincon`, `fminunc`, `lsqcurvefit`, or `lsqnonlin`. The `GlobalSearch` algorithm uses `fmincon`.
- `MultiStart` can run in parallel, distributing start points to multiple processors for local solution. To run `MultiStart` in parallel, see “How to Use Parallel Processing” on page 9-12.

Deciding Which Solver to Use

The differences between these solver objects boil down to the following decision on which to use:

- Use `GlobalSearch` to find a single global minimum most efficiently on a single processor.
- Use `MultiStart` to:
  - Find multiple local minima.
  - Run in parallel.
  - Use a solver other than `fmincon`.
  - Search thoroughly for a global minimum.
  - Explore your own start points.

GlobalSearch Algorithm

For a description of the algorithm, see Ugray et al. [1].

When you `run` a `GlobalSearch` object, the algorithm performs the following steps:

1. “Run fmincon from x0” on page 3-46
2. “Generate Trial Points” on page 3-47
3. “Obtain Stage 1 Start Point, Run” on page 3-47
4. “Initialize Basins, Counters, Threshold” on page 3-47
5. “Begin Main Loop” on page 3-48
6. “Examine Stage 2 Trial Point to See if fmincon Runs” on page 3-48
7. “When fmincon Runs” on page 3-48
8. “When fmincon Does Not Run” on page 3-49
9. “Create GlobalOptimSolution” on page 3-50

Run fmincon from x0

`GlobalSearch` runs `fmincon` from the start point you give in the `problem` structure. If this run converges, `GlobalSearch` records the start point and end point for an initial estimate on the radius of a basin of attraction. Furthermore, `GlobalSearch` records the final objective function value for use in the *score* function (see “Obtain Stage 1 Start Point, Run” on page 3-47).

The score function is the sum of the objective function value at a point and a multiple of the sum of the constraint violations. So a feasible point has score equal to its objective function value. The multiple for constraint violations is initially 1000. `GlobalSearch` updates the multiple during the run.

Generate Trial Points

`GlobalSearch` uses the scatter search algorithm to generate a set of `NumTrialPoints` trial points. Trial points are potential start points. For a description of the scatter search algorithm, see Glover [2]. `GlobalSearch` generates trial points within any finite bounds you set (`lb` and `ub`). Unbounded components have artificial bounds imposed: `lb = -1e4 + 1`, `ub = 1e4 + 1`. This range is not symmetric about the origin so that the origin is not in the scatter search. Components with one-sided bounds have artificial bounds imposed on the unbounded side, shifted by the finite bounds to keep `lb < ub`.

Obtain Stage 1 Start Point, Run

`GlobalSearch` evaluates the score function of a set of `NumStageOnePoints` trial points. It then takes the point with the best score and runs `fmincon` from that point. `GlobalSearch` removes the set of `NumStageOnePoints` trial points from its list of points to examine.

Initialize Basins, Counters, Threshold

The `localSolverThreshold` is initially the smaller of the two objective function values at the solution points. The solution points are the `fmincon` solutions starting from `x0` and from the Stage 1 start point. If both of these solution points do not exist or are infeasible, `localSolverThreshold` is initially the penalty function value of the Stage 1 start point.

The `GlobalSearch` heuristic assumption is that basins of attraction are spherical. The initial estimate of basins of attraction for the solution point from `x0` and the solution point from Stage 1 are spheres centered at the solution points. The radius of each sphere is the distance from the initial point to the solution point. These estimated basins can overlap.

There are two sets of counters associated with the algorithm. Each counter is the number of consecutive trial points that:

- Lie within a basin of attraction. There is one counter for each basin.
- Have score function greater than `localSolverThreshold`. For a definition of the score, see “Run fmincon from x0” on page 3-46.

All counters are initially 0.

Begin Main Loop

`GlobalSearch` repeatedly examines a remaining trial point from the list, and performs the following steps. It continually monitors the time, and stops the search if elapsed time exceeds `MaxTime` seconds.

Examine Stage 2 Trial Point to See if fmincon Runs

Call the trial point `p`. Run `fmincon` from `p` if the following conditions hold:

- `p` is not in any existing basin. The criterion for every basin `i` is:

  ```
  |p - center(i)| > DistanceThresholdFactor * radius(i).
  ```

  `DistanceThresholdFactor` is an option (default value `0.75`).

  `radius` is an estimated radius that updates in Update Basin Radius and Threshold and React to Large Counter Values.
- score(`p`) < `localSolverThreshold`.
- (optional) `p` satisfies bound and/or inequality constraints. This test occurs if you set the `StartPointsToRun` property of the `GlobalSearch` object to `'bounds'` or `'bounds-ineqs'`.

When fmincon Runs

1. **Reset Counters**

   Set the counters for basins and threshold to 0.
2. **Update Solution Set**

   If `fmincon` runs starting from `p`, it can yield a positive exit flag, which indicates convergence. In that case, `GlobalSearch` updates the vector of `GlobalOptimSolution` objects. Call the solution point `xp` and the objective function value `fp`. There are two cases:

   - For every other solution point `xq` with objective function value `fq`,

     ```
     |xq - xp| > TolX * max(1,|xp|)
     ```

     or

     ```
     |fq - fp| > TolFun * max(1,|fp|).
     ```

In this case, `GlobalSearch` creates a new element in the vector of `GlobalOptimSolution` objects. For details of the information contained in each object, see `GlobalOptimSolution`.

- For some other solution point `xq` with objective function value `fq`,

  ```
  |xq - xp| <= TolX * max(1,|xp|)
  ```

  and

  ```
  |fq - fp| <= TolFun * max(1,|fp|).
  ```

  In this case, `GlobalSearch` regards `xp` as equivalent to `xq`. The `GlobalSearch` algorithm modifies the `GlobalOptimSolution` of `xq` by adding `p` to the cell array of `X0` points.

  There is one minor tweak that can happen to this update. If the exit flag for `xq` is greater than `1`, and the exit flag for `xp` is `1`, then `xp` replaces `xq`. This replacement can lead to some points in the same basin being more than a distance of `TolX` from `xp`.

3 **Update Basin Radius and Threshold**

   If the exit flag of the current `fmincon` run is positive:

   a Set threshold to the score value at start point `p`.
   b Set basin radius for `xp` equal to the maximum of the existing radius (if any) and the distance between `p` and `xp`.

4 **Report to Iterative Display**

   When the `GlobalSearch` `Display` property is `'iter'`, every point that `fmincon` runs creates one line in the `GlobalSearch` iterative display.

When fmincon Does Not Run

1 **Update Counters**

   Increment the counter for every basin containing `p`. Reset the counter of every other basin to `0`.

   Increment the threshold counter if score(`p`) >= `localSolverThreshold`. Otherwise, reset the counter to `0`.

2 **React to Large Counter Values**

For each basin with counter equal to `MaxWaitCycle`, multiply the basin radius by 1 – `BasinRadiusFactor`. Reset the counter to `0`. (Both `MaxWaitCycle` and `BasinRadiusFactor` are settable properties of the `GlobalSearch` object.)

If the threshold counter equals `MaxWaitCycle`, increase the threshold: new threshold = threshold + `PenaltyThresholdFactor`*(1 + abs(threshold)).

Reset the counter to `0`.

3 **Report to Iterative Display**

Every 200th trial point creates one line in the `GlobalSearch` iterative display.

Create GlobalOptimSolution

After reaching `MaxTime` seconds or running out of trial points, `GlobalSearch` creates a vector of `GlobalOptimSolution` objects. `GlobalSearch` orders the vector by objective function value, from lowest (best) to highest (worst). This concludes the algorithm.

MultiStart Algorithm

When you `run` a `MultiStart` object, the algorithm performs the following steps:

- "Generate Start Points" on page 3-50
- "Filter Start Points (Optional)" on page 3-51
- "Run Local Solver" on page 3-51
- "Check Stopping Conditions" on page 3-51
- "Create GlobalOptimSolution Object" on page 3-51

Generate Start Points

If you call `MultiStart` with the syntax

```
[x fval] = run(ms,problem,k)
```

for an integer `k`, `MultiStart` generates `k - 1` start points exactly as if you used a `RandomStartPointSet` object. The algorithm also uses the `x0` start point from the `problem` structure, for a total of `k` start points.

A `RandomStartPointSet` object does not have any points stored inside the object. Instead, `MultiStart` calls the `list` method, which generates random points within the bounds given by the `problem` structure. If an unbounded component exists, `list` uses an artificial bound given by the `ArtificialBound` property of the `RandomStartPointSet` object.

If you provide a `CustomStartPointSet` object, `MultiStart` does not generate start points, but uses the points in the object.

Filter Start Points (Optional)

If you set the `StartPointsToRun` property of the `MultiStart` object to `'bounds'` or `'bounds-ineqs'`, `MultiStart` does not run the local solver from infeasible start points. In this context, "infeasible" means start points that do not satisfy bounds, or start points that do not satisfy both bounds and inequality constraints.

The default setting of `StartPointsToRun` is `'all'`. In this case, `MultiStart` does not discard infeasible start points.

Run Local Solver

`MultiStart` runs the local solver specified in `problem.solver`, starting at the points that pass the `StartPointsToRun` filter. If `MultiStart` is running in parallel, it sends start points to worker processors one at a time, and the worker processors run the local solver.

The local solver checks whether `MaxTime` seconds have elapsed at each of its iterations. If so, it exits that iteration without reporting a solution.

When the local solver stops, `MultiStart` stores the results and continues to the next step.

Report to Iterative Display

When the `MultiStart` `Display` property is `'iter'`, every point that the local solver runs creates one line in the `MultiStart` iterative display.

Check Stopping Conditions

`MultiStart` stops when it runs out of start points. It also stops when it exceeds a total run time of `MaxTime` seconds.

Create GlobalOptimSolution Object

After `MultiStart` reaches a stopping condition, the algorithm creates a vector of `GlobalOptimSolution` objects as follows:

1. Sort the local solutions by objective function value (`Fval`) from lowest to highest. For the `lsqnonlin` and `lsqcurvefit` local solvers, the objective function is the norm of the residual.
2. Loop over the local solutions `j` beginning with the lowest (best) `Fval`.
3. Find all the solutions `k` satisfying both:

   ```
   |Fval(k) - Fval(j)| <= TolFun*max(1,|Fval(j)|)
   ```

   ```
   |x(k) - x(j)| <= TolX*max(1,|x(j)|)
   ```
4. Record `j`, `Fval(j)`, the local solver `output` structure for `j`, and a cell array of the start points for `j` and all the `k`. Remove those points `k` from the list of local solutions. This point is one entry in the vector of `GlobalOptimSolution` objects.

The resulting vector of `GlobalOptimSolution` objects is in order by `Fval`, from lowest (best) to highest (worst).

Report to Iterative Display

After examining all the local solutions, `MultiStart` gives a summary to the iterative display. This summary includes the number of local solver runs that converged, the number that failed to converge, and the number that had errors.

Bibliography

[1] Ugray, Zsolt, Leon Lasdon, John C. Plummer, Fred Glover, James Kelly, and Rafael Martí. *Scatter Search and Local NLP Solvers: A Multistart Framework for Global Optimization*. INFORMS Journal on Computing, Vol. 19, No. 3, 2007, pp. 328–340.

[2] Glover, F. “A template for scatter search and path relinking.” *Artificial Evolution* (J.-K. Hao, E.Lutton, E.Ronald, M.Schoenauer, D.Snyers, eds.). Lecture Notes in Computer Science, 1363, Springer, Berlin/Heidelberg, 1998, pp. 13–54.

[3] Dixon, L. and G. P. Szegö. “The Global Optimization Problem: an Introduction.” *Towards Global Optimisation 2* (Dixon, L. C. W. and G. P. Szegö, eds.). Amsterdam, The Netherlands: North Holland, 1978.

Can You Certify a Solution Is Global?

In this section...
"No Guarantees" on page 3-53
"Check if a Solution Is a Local Solution with patternsearch" on page 3-53
"Identify a Bounded Region That Contains a Global Solution" on page 3-54
"Use MultiStart with More Start Points" on page 3-55

No Guarantees

How can you tell if you have located the global minimum of your objective function? The short answer is that you cannot; you have no guarantee that the result of a Global Optimization Toolbox solver is a global optimum.

However, you can use the strategies in this section for investigating solutions.

Check if a Solution Is a Local Solution with patternsearch

Before you can determine if a purported solution is a global minimum, first check that it is a local minimum. To do so, run `patternsearch` on the problem.

To convert the `problem` to use `patternsearch` instead of `fmincon` or `fminunc`, enter

```
problem.solver = 'patternsearch';
```

Also, change the start point to the solution you just found, and clear the options:

```
problem.x0 = x;
problem.options = [];
```

For example, Check Nearby Points (in the Optimization Toolbox documentation) shows the following:

```
options = optimoptions(@fmincon,'Algorithm','active-set');
ffun = @(x)(x(1)-(x(1)-x(2))^2);
problem = createOptimProblem('fmincon', ...
    'objective',ffun,'x0',[1/2 1/3], ...
    'lb',[0 -1],'ub',[1 1],'options',options);
[x fval exitflag] = fmincon(problem)
```

```
x =
  1.0e-007 *
         0    0.1614

fval =
 -2.6059e-016

exitflag =
     1
```

However, checking this purported solution with `patternsearch` shows that there is a better solution. Start `patternsearch` from the reported solution `x`:

```
% set the candidate solution x as the start point
problem.x0 = x;
problem.solver = 'patternsearch';
problem.options = [];
[xp fvalp exitflagp] = patternsearch(problem)

Optimization terminated: mesh size less than options.TolMesh.

xp =

    1.0000   -1.0000


fvalp =

   -3.0000


exitflagp =

     1
```

Identify a Bounded Region That Contains a Global Solution

Suppose you have a smooth objective function in a bounded region. Given enough time and start points, `MultiStart` eventually locates a global solution.

Therefore, if you can bound the region where a global solution can exist, you can obtain some degree of assurance that `MultiStart` locates the global solution.

For example, consider the function

$$f = x^6 + y^6 + \sin(x+y)\left(x^2+y^2\right) - \cos\left(\frac{x^2}{1+y^2}\right)\left(2+x^4+x^2y^2+y^4\right).$$

The initial summands $x^6 + y^6$ force the function to become large and positive for large values of $|x|$ or $|y|$. The components of the global minimum of the function must be within the bounds
$-10 \le x,y \le 10$,

since 10^6 is much larger than all the multiples of 10^4 that occur in the other summands of the function.

You can identify smaller bounds for this problem; for example, the global minimum is between –2 and 2. It is more important to identify reasonable bounds than it is to identify the best bounds.

Use MultiStart with More Start Points

To check whether there is a better solution to your problem, run `MultiStart` with additional start points. Use `MultiStart` instead of `GlobalSearch` for this task because `GlobalSearch` does not run the local solver from all start points.

For example, see “Example: Searching for a Better Solution” on page 3-60.

Updating Unconstrained Problem from GlobalSearch

If you use `GlobalSearch` on an unconstrained problem, change your `problem` structure before using `MultiStart`. You have two choices in updating a `problem` structure for an unconstrained problem using `MultiStart`:

- Change the `solver` field to `'fminunc'`:

  ```
  problem.solver = 'fminunc';
  ```

  To avoid a warning if your objective function does not compute a gradient, change the local `options` to have `Algorithm` set to `'quasi-newton'`:

  ```
  problem.options.Algorithm = 'quasi-newton';
  ```

- Add an artificial constraint, retaining `fmincon` as the local solver:

  ```
  problem.lb = -Inf(size(x0));
  ```

To search a larger region than the default, see “Refine Start Points” on page 3-57.

Refine Start Points

In this section...

About Refining Start Points

If some components of your problem are unconstrained, `GlobalSearch` and `MultiStart` use artificial bounds to generate random start points uniformly in each component. However, if your problem has far-flung minima, you need widely dispersed start points to find these minima.

Use these methods to obtain widely dispersed start points:

- Give widely separated bounds in your `problem` structure.
- Use a `RandomStartPointSet` object with the `MultiStart` algorithm. Set a large value of the `ArtificialBound` property in the `RandomStartPointSet` object.
- Use a `CustomStartPointSet` object with the `MultiStart` algorithm. Use widely dispersed start points.

There are advantages and disadvantages of each method.

Method	Advantages	Disadvantages
Give bounds in `problem`	Automatic point generation	Makes a more complex Hessian
	Can use with `GlobalSearch`	Unclear how large to set the bounds
	Easy to do	Changes `problem`
	Bounds can be asymmetric	Only uniform points
Large `ArtificialBound` in `RandomStartPointSet`	Automatic point generation	`MultiStart` only
	Does not change `problem`	Only symmetric, uniform points
	Easy to do	Unclear how large to set `ArtificialBound`

Method	Advantages	Disadvantages
`CustomStartPointSet`	Customizable	`MultiStart` only
	Does not change `problem`	Requires programming for generating points

Methods of Generating Start Points

- "Uniform Grid" on page 3-58
- "Perturbed Grid" on page 3-59
- "Widely Dispersed Points for Unconstrained Components" on page 3-59

Uniform Grid

To generate a uniform grid of start points:

1. Generate multidimensional arrays with `ndgrid`. Give the lower bound, spacing, and upper bound for each component.

   For example, to generate a set of three-dimensional arrays with

   - First component from –2 through 0, spacing 0.5
   - Second component from 0 through 2, spacing 0.25
   - Third component from –10 through 5, spacing 1

   ```
   [X,Y,Z] = ndgrid(-2:.5:0,0:.25:2,-10:5);
   ```
2. Place the arrays into a single matrix, with each row representing one start point. For example:

   ```
   W = [X(:),Y(:),Z(:)];
   ```

   In this example, `W` is a 720-by-3 matrix.
3. Put the matrix into a `CustomStartPointSet` object. For example:

   ```
   custpts = CustomStartPointSet(W);
   ```

Call `MultiStart run` with the `CustomStartPointSet` object as the third input. For example,

```
% Assume problem structure and ms MultiStart object exist
[x fval flag outpt manymins] = run(ms,problem,custpts);
```

Perturbed Grid

Integer start points can yield less robust solutions than slightly perturbed start points.

To obtain a perturbed set of start points:

1 Generate a matrix of start points as in steps 1–2 of "Uniform Grid" on page 3-58.
2 Perturb the start points by adding a random normal matrix with 0 mean and relatively small variance.

   For the example in "Uniform Grid" on page 3-58, after making the `W` matrix, add a perturbation:

```
[X,Y,Z] = ndgrid(-2:.5:0,0:.25:2,-10:5);
W = [X(:),Y(:),Z(:)];
W = W + 0.01*randn(size(W));
```
3 Put the matrix into a `CustomStartPointSet` object. For example:

```
custpts = CustomStartPointSet(W);
```

Call `MultiStart run` with the `CustomStartPointSet` object as the third input. For example,

```
% Assume problem structure and ms MultiStart object exist
[x fval flag outpt manymins] = run(ms,problem,custpts);
```

Widely Dispersed Points for Unconstrained Components

Some components of your problem can lack upper or lower bounds. For example:

- Although no explicit bounds exist, there are levels that the components cannot attain. For example, if one component represents the weight of a single diamond, there is an implicit upper bound of 1 kg (the Hope Diamond is under 10 g). In such a case, give the implicit bound as an upper bound.
- There truly is no upper bound. For example, the size of a computer file in bytes has no effective upper bound. The largest size can be in gigabytes or terabytes today, but in 10 years, who knows?

For truly unbounded components, you can use the following methods of sampling. To generate approximately $1/n$ points in each region $(\exp(n),\exp(n+1))$, use the following formula. If u is random and uniformly distributed from 0 through 1, then $r = 2u – 1$ is uniformly distributed between –1 and 1. Take

$$y = \text{sgn}(r)\left(\exp(1/|r|) - e\right).$$

y is symmetric and random. For a variable bounded below by `lb`, take

$$y = \text{lb} + \left(\exp(1/u) - e\right).$$

Similarly, for a variable bounded above by `ub`, take

$$y = \text{ub} - \left(\exp(1/u) - e\right).$$

For example, suppose you have a three-dimensional problem with

- `x`(1) > 0
- `x`(2) < 100
- `x`(3) unconstrained

To make 150 start points satisfying these constraints:

```
u = rand(150,3);
r1 = 1./u(:,1);
r1 = exp(r1) - exp(1);
r2 = 1./u(:,2);
r2 = -exp(r2) + exp(1) + 100;
r3 = 1./(2*u(:,3)-1);
r3 = sign(r3).*(exp(abs(r3)) - exp(1));
custpts = CustomStartPointSet([r1,r2,r3]);
```

The following is a variant of this algorithm. Generate a number between 0 and infinity by the method for lower bounds. Use this number as the radius of a point. Generate the other components of the point by taking random numbers for each component and multiply by the radius. You can normalize the random numbers, before multiplying by the radius, so their norm is 1. For a worked example of this method, see “MultiStart Without Bounds, Widely Dispersed Start Points” on page 3-95.

Example: Searching for a Better Solution

`MultiStart` fails to find the global minimum in “Multiple Local Minima Via MultiStart” on page 3-76. There are two simple ways to search for a better solution:

- Use more start points
- Give tighter bounds on the search space

Set up the problem structure and `MultiStart` object:

```
problem = createOptimProblem('fminunc',...
    'objective',@(x)sawtoothxy(x(1),x(2)),...
    'x0',[100,-50],'options',...
    optimoptions(@fminunc,'Algorithm','quasi-newton'));
ms = MultiStart;
```

Use More Start Points

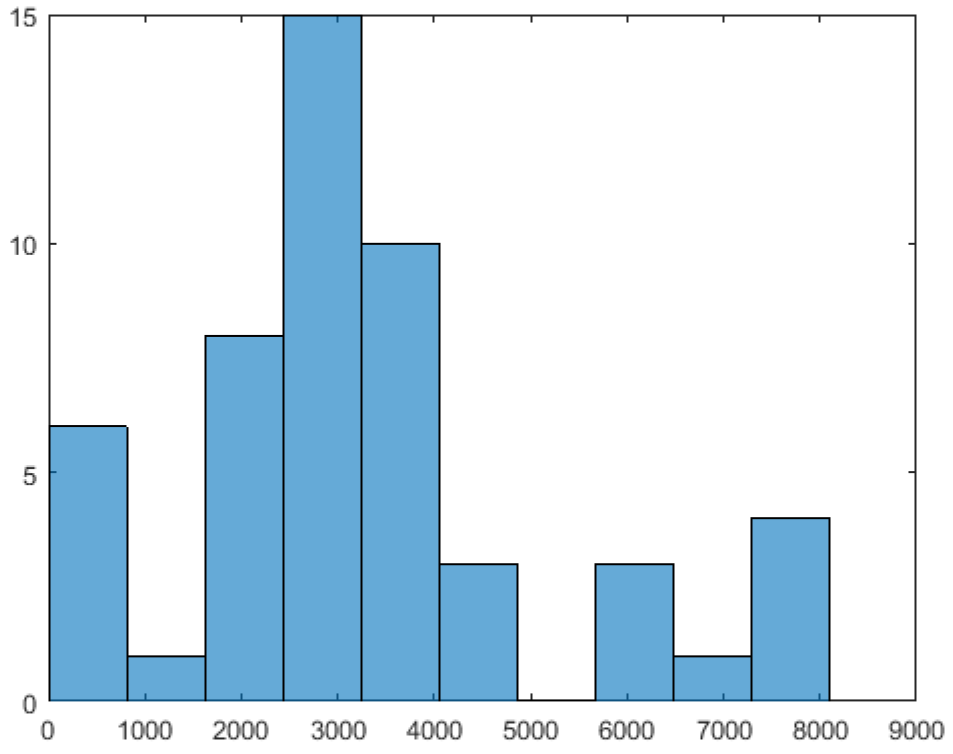
Run `MultiStart` on the problem for 200 start points instead of 50:

```
rng(14,'twister') % for reproducibility
[x fval eflag output manymins] = run(ms,problem,200)

MultiStart completed some of the runs from the start points.

51 out of 200 local solver runs converged with a positive local solver exit flag.

x =

   1.0e-06 *

   -0.2284   -0.5567


fval =

   2.1382e-12


eflag =

     2


output =

                funcCount: 32760
         localSolverTotal: 200
       localSolverSuccess: 51
    localSolverIncomplete: 149
    localSolverNoSolution: 0
                  message: 'MultiStart completed some of the runs from the start poin..
```

```
manymins =
  1x51 GlobalOptimSolution

  Properties:
    X
    Fval
    Exitflag
    Output
    X0
```

This time `MultiStart` found the global minimum, and found 51 local minima.

To see the range of local solutions, enter `histogram([manymins.Fval],10)`.

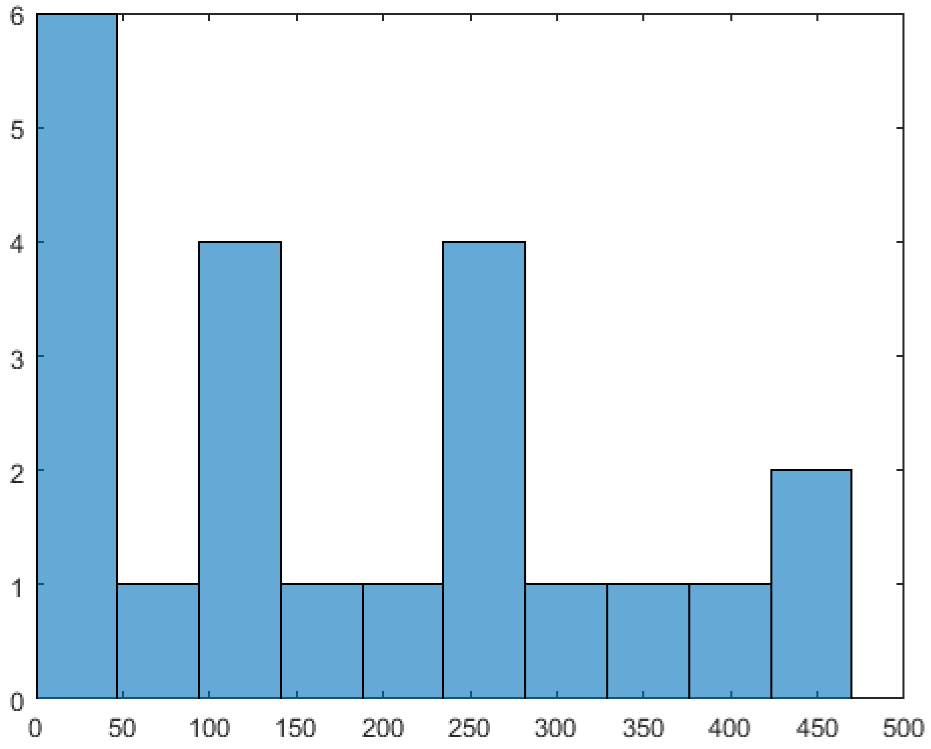


Tighter Bound on the Start Points

Suppose you believe that the interesting local solutions have absolute values of all components less than 100. The default value of the bound on start points is 1000. To use a different value of the bound, generate a `RandomStartPointSet` with the `ArtificialBound` property set to `100`:

```
startpts = RandomStartPointSet('ArtificialBound',100,...
    'NumStartPoints',50);
[x fval eflag output manymins] = run(ms,problem,startpts)

MultiStart completed some of the runs from the start points.

27 out of 50 local solver runs converged with a positive local solver exit flag.

x =

   1.0e-08 *

    0.9725   -0.6198


fval =

   1.4955e-15


eflag =

     2


output =

                funcCount: 7482
         localSolverTotal: 50
       localSolverSuccess: 27
    localSolverIncomplete: 23
    localSolverNoSolution: 0
                  message: 'MultiStart completed some of the runs from the start poin

manymins =
  1x22 GlobalOptimSolution
  Properties:
    X
    Fval
    Exitflag
    Output
    X0
```

`MultiStart` found the global minimum, and found 22 distinct local solutions. To see the range of local solutions, enter `histogram([manymins.Fval],10)`.



Compared to the minima found in "Use More Start Points" on page 3-61, this run found better (smaller) minima, and had a higher percentage of successful runs.

Change Options

In this section...

How to Determine Which Options to Change

After you run a global solver, you might want to change some global or local options. To determine which options to change, the guiding principle is:

- To affect the local solver, set local solver options.
- To affect the start points or solution set, change the `problem` structure, or set the global solver object properties.

For example, to obtain:

- More local minima — Set global solver object properties.
- Faster local solver iterations — Set local solver options.
- Different tolerances for considering local solutions identical (to obtain more or fewer local solutions) — Set global solver object properties.
- Different information displayed at the command line — Decide if you want iterative display from the local solver (set local solver options) or global information (set global solver object properties).
- Different bounds, to examine different regions — Set the bounds in the `problem` structure.

Examples of Choosing Problem Options

- To start your local solver at points only satisfying inequality constraints, set the `StartPointsToRun` property in the global solver object to `'bounds-ineqs'`. This setting can speed your solution, since local solvers do not have to attempt to find points satisfying these constraints. However, the setting can result in many fewer local solver runs, since the global solver can reject many start points. For an example, see "Optimize Using Only Feasible Start Points" on page 3-80.

- To use the `fmincon interior-point` algorithm, set the local solver `Algorithm` option to `'interior-point'`. For an example showing how to do this, see "Examples of Updating Problem Options" on page 3-66.
- For your local solver to have different bounds, set the bounds in the `problem` structure. Examine different regions by setting bounds.
- To see every solution that has positive local exit flag, set the `TolX` property in the global solver object to `0`. For an example showing how to do this, see "Changing Global Options" on page 3-67.

Changing Local Solver Options

There are several ways to change values in a local options structure:

- Update the values using dot notation and `optimoptions`. The syntax is

  ```
  problem.options = optimoptions(problem.options,'Parameter',value,...);
  ```

  You can also replace the local options entirely:

  ```
  problem.options = optimoptions(@solvername,'Parameter',value,...);
  ```
- Use dot notation on one local option. The syntax is

  ```
  problem.options.Parameter = newvalue;
  ```
- Recreate the entire problem structure. For details, see "Create Problem Structure" on page 3-6.

Examples of Updating Problem Options

1. Create a problem structure:

   ```
   problem = createOptimProblem('fmincon','x0',[-1 2], ...
       'objective',@rosenboth);
   ```
2. Set the problem to use the `sqp` algorithm in `fmincon`:

   ```
   problem.options.Algorithm = 'sqp';
   ```
3. Update the problem to use the gradient in the objective function, have a `TolFun` value of `1e-8`, and a `TolX` value of `1e-7`:

   ```
   problem.options = optimoptions(problem.options,'GradObj','on', ...
       'TolFun',1e-8,'TolX',1e-7);
   ```

Changing Global Options

There are several ways to change characteristics of a `GlobalSearch` or `MultiStart` object:

- Use dot notation. For example, suppose you have a default `MultiStart` object:

  ```
  ms = MultiStart

    MultiStart with properties:
           UseParallel: 0
               Display: 'final'
                TolFun: 1.0000e-06
                  TolX: 1.0000e-06
               MaxTime: Inf
      StartPointsToRun: 'all'
            OutputFcns: []
              PlotFcns: []
  ```

  To change `ms` to have its `TolX` value equal to `1e-3`, update the `TolX` field:

  ```
  ms.TolX = 1e-3

    MultiStart with properties:
           UseParallel: 0
               Display: 'final'
                TolFun: 1.0000e-06
                  TolX: 1.0000e-03
               MaxTime: Inf
      StartPointsToRun: 'all'
            OutputFcns: []
              PlotFcns: []
  ```

- Reconstruct the object starting from the current settings. For example, to set the `TolFun` field in `ms` to `1e-3`, retaining the nondefault value for `TolX`:

  ```
  ms = MultiStart(ms,'TolFun',1e-3)

    MultiStart with properties:
           UseParallel: 0
               Display: 'final'
                TolFun: 1.0000e-03
                  TolX: 1.0000e-03
               MaxTime: Inf
      StartPointsToRun: 'all'
  ```

```
          OutputFcns: []
            PlotFcns: []
```

- Convert a `GlobalSearch` object to a `MultiStart` object, or vice-versa. For example, with the `ms` object from the previous example, create a `GlobalSearch` object with the same values of `TolX` and `TolFun`:

  ```
  gs = GlobalSearch(ms)

    GlobalSearch with properties:
               NumTrialPoints: 1000
            BasinRadiusFactor: 0.2000
      DistanceThresholdFactor: 0.7500
                 MaxWaitCycle: 20
            NumStageOnePoints: 200
       PenaltyThresholdFactor: 0.2000
                      Display: 'final'
                       TolFun: 1.0000e-03
                         TolX: 1.0000e-03
                      MaxTime: Inf
             StartPointsToRun: 'all'
                   OutputFcns: []
                     PlotFcns: []
  ```

Reproduce Results

In this section...

Identical Answers with Pseudorandom Numbers

`GlobalSearch` and `MultiStart` use pseudorandom numbers in choosing start points. Use the same pseudorandom number stream again to:

- Compare various algorithm settings.
- Have an example run repeatably.
- Extend a run, with known initial segment of a previous run.

Both `GlobalSearch` and `MultiStart` use the default random number stream.

Steps to Take in Reproducing Results

1. Before running your problem, store the current state of the default random number stream:

   ```
   stream = rng;
   ```
2. Run your `GlobalSearch` or `MultiStart` problem.
3. Restore the state of the random number stream:

   ```
   rng(stream)
   ```
4. If you run your problem again, you get the same result.

Example: Reproducing a GlobalSearch or MultiStart Result

This example shows how to obtain reproducible results for "Find Global or Multiple Local Minima" on page 3-72. The example follows the procedure in "Steps to Take in Reproducing Results" on page 3-69.

1. Store the current state of the default random number stream:

```
stream = rng;
```

2 Create the `sawtoothxy` function file:

```
function f = sawtoothxy(x,y)
[t r] = cart2pol(x,y); % change to polar coordinates
h = cos(2*t - 1/2)/2 + cos(t) + 2;
g = (sin(r) - sin(2*r)/2 + sin(3*r)/3 - sin(4*r)/4 + 4) ...
    .*r.^2./(r+1);
f = g.*h;
end
```

3 Create the `problem` structure and `GlobalSearch` object:

```
problem = createOptimProblem('fmincon',...
    'objective',@(x)sawtoothxy(x(1),x(2)),...
    'x0',[100,-50],'options',...
    optimoptions(@fmincon,'Algorithm','sqp'));
gs = GlobalSearch('Display','iter');
```

4 Run the problem:

```
[x fval] = run(gs,problem)

 Num Pts                 Best       Current    Threshold        Local        Local
Analyzed  F-count        f(x)       Penalty      Penalty         f(x)     exitflag        Procedure
       0      465       422.9                                   422.9            2    Initial Point
     200     1730   1.547e-015                               1.547e-015          1    Stage 1 Local
     300     1830   1.547e-015     6.01e+004        1.074                              Stage 2 Search
     400     1930   1.547e-015     1.47e+005         4.16                              Stage 2 Search
     500     2030   1.547e-015     2.63e+004        11.84                              Stage 2 Search
     600     2130   1.547e-015    1.341e+004        30.95                              Stage 2 Search
     700     2230   1.547e-015    2.562e+004        65.25                              Stage 2 Search
     800     2330   1.547e-015    5.217e+004        163.8                              Stage 2 Search
     900     2430   1.547e-015    7.704e+004        409.2                              Stage 2 Search
     981     2587   1.547e-015         42.24        516.6        7.573            1     Stage 2 Local
    1000     2606   1.547e-015    3.299e+004        42.24                              Stage 2 Search

GlobalSearch stopped because it analyzed all the trial points.

All 3 local solver runs converged with a positive local solver exit flag.

x =
  1.0e-007 *
    0.0414    0.1298

fval =
  1.5467e-015
```

You might obtain a different result when running this problem, since the random stream was in an unknown state at the beginning of the run.

5 Restore the state of the random number stream:

```
rng(stream)
```

6 Run the problem again. You get the same output.

```
[x fval] = run(gs,problem)

 Num Pts                 Best       Current    Threshold        Local        Local
Analyzed  F-count        f(x)       Penalty      Penalty         f(x)     exitflag        Procedure
       0      465       422.9                                   422.9            2    Initial Point
     200     1730  1.547e-015                              1.547e-015            1    Stage 1 Local

... Output deleted to save space ...

x =
  1.0e-007 *
    0.0414    0.1298

fval =
  1.5467e-015
```

Parallel Processing and Random Number Streams

You obtain reproducible results from `MultiStart` when you run the algorithm in parallel the same way as you do for serial computation. Runs are reproducible because `MultiStart` generates pseudorandom start points locally, and then distributes the start points to parallel processors. Therefore, the parallel processors do not use random numbers.

To reproduce a parallel `MultiStart` run, use the procedure described in “Steps to Take in Reproducing Results” on page 3-69. For a description of how to run `MultiStart` in parallel, see “How to Use Parallel Processing” on page 9-12.

Find Global or Multiple Local Minima

In this section...
"Function to Optimize" on page 3-72
"Single Global Minimum Via GlobalSearch" on page 3-74
"Multiple Local Minima Via MultiStart" on page 3-76

Function to Optimize

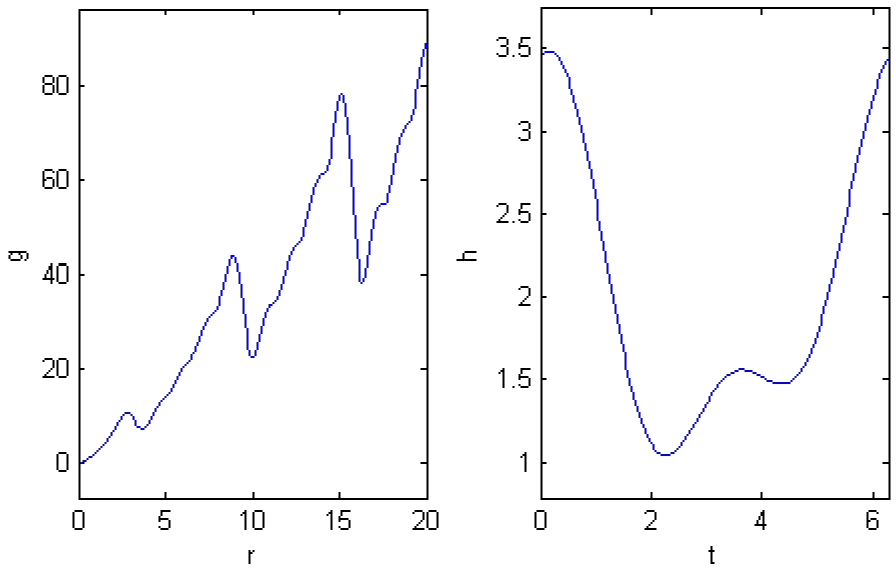
This example illustrates how `GlobalSearch` finds a global minimum efficiently, and how `MultiStart` finds many more local minima.

The objective function for this example has many local minima and a unique global minimum. In polar coordinates, the function is
$f(r,t) = g(r)h(t)$,

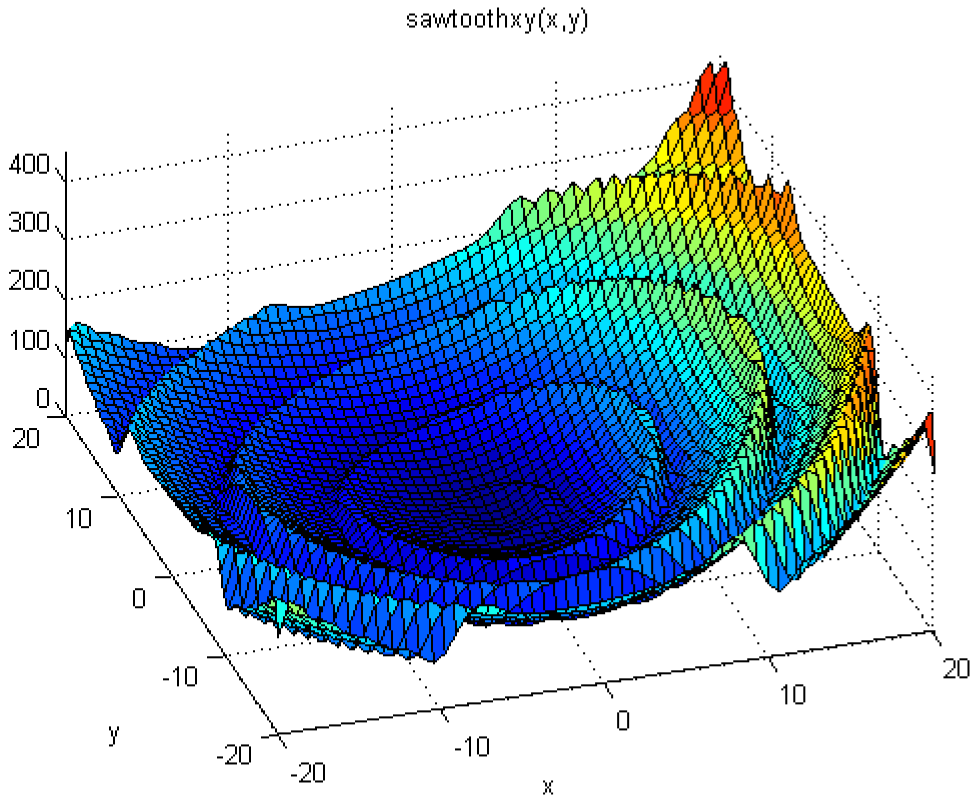
where

$$g(r) = \left(\sin(r) - \frac{\sin(2r)}{2} + \frac{\sin(3r)}{3} - \frac{\sin(4r)}{4} + 4\right)\frac{r^2}{r+1}$$

$$h(t) = 2 + \cos(t) + \frac{\cos\left(2t - \frac{1}{2}\right)}{2}.$$



The global minimum is at $r = 0$, with objective function 0. The function $g(r)$ grows approximately linearly in r, with a repeating sawtooth shape. The function $h(t)$ has two local minima, one of which is global.



Single Global Minimum Via GlobalSearch

1 Write a function file to compute the objective:

```
function f = sawtoothxy(x,y)
[t r] = cart2pol(x,y); % change to polar coordinates
h = cos(2*t - 1/2)/2 + cos(t) + 2;
g = (sin(r) - sin(2*r)/2 + sin(3*r)/3 - sin(4*r)/4 + 4) ...
    .*r.^2./(r+1);
f = g.*h;
end
```

2 Create the problem structure. Use the `'sqp'` algorithm for `fmincon`:

```
problem = createOptimProblem('fmincon',...
    'objective',@(x)sawtoothxy(x(1),x(2)),...
    'x0',[100,-50],'options',...
    optimoptions(@fmincon,'Algorithm','sqp','Display','off'));
```

The start point is `[100,-50]` instead of `[0,0]`, so `GlobalSearch` does not start at the global solution.

3 Add an artificial bound, and validate the problem structure by running `fmincon`:

```
problem.lb = -Inf([2,1]);
[x fval] = fmincon(problem)

x =

   45.6965 -107.6645


fval =

  555.6941
```

4 Create the `GlobalSearch` object, and set iterative display:

```
gs = GlobalSearch('Display','iter');
```

5 Run the solver:

```
rng(14,'twister') % for reproducibility
[x fval] = run(gs,problem)

 Num Pts                 Best       Current    Threshold        Local        Local                 
Analyzed  F-count        f(x)       Penalty      Penalty         f(x)     exitflag        Procedure
       0      200       555.7                                   555.7            0    Initial Point
     200     1479   1.547e-15                               1.547e-15            1    Stage 1 Local
     300     1580   1.547e-15     5.858e+04        1.074                              Stage 2 Search
     400     1680   1.547e-15      1.84e+05         4.16                              Stage 2 Search
     500     1780   1.547e-15     2.683e+04        11.84                              Stage 2 Search
     600     1880   1.547e-15     1.122e+04        30.95                              Stage 2 Search
     700     1980   1.547e-15     1.353e+04        65.25                              Stage 2 Search
     800     2080   1.547e-15     6.249e+04        163.8                              Stage 2 Search
     900     2180   1.547e-15     4.119e+04        409.2                              Stage 2 Search
     950     2372   1.547e-15           477        589.7          387            2    Stage 2 Local
     952     2436   1.547e-15         368.4          477        250.7            2    Stage 2 Local
    1000     2484   1.547e-15     4.031e+04        530.9                              Stage 2 Search

GlobalSearch stopped because it analyzed all the trial points.

3 out of 4 local solver runs converged with a positive local solver exit flag.

x =

   1.0e-07 *
```

```
       0.0414    0.1298

    fval =

      1.5467e-15
```

You can get different results, since `GlobalSearch` is stochastic.

The solver found three local minima, and it found the global minimum near `[0,0]`.

Multiple Local Minima Via MultiStart

1 Write a function file to compute the objective:

```
function f = sawtoothxy(x,y)
[t r] = cart2pol(x,y); % change to polar coordinates
h = cos(2*t - 1/2)/2 + cos(t) + 2;
g = (sin(r) - sin(2*r)/2 + sin(3*r)/3 - sin(4*r)/4 + 4) ...
    .*r.^2./(r+1);
f = g.*h;
end
```

2 Create the problem structure. Use the `fminunc` solver with the `Algorithm` option set to `'quasi-newton'`. The reasons for these choices are:

- The problem is unconstrained. Therefore, `fminunc` is the appropriate solver; see "Optimization Decision Table" in the Optimization Toolbox documentation.
- The default `fminunc` algorithm requires a gradient; see "Choosing the Algorithm" in the Optimization Toolbox documentation. Therefore, set `Algorithm` to `'quasi-newton'`.

```
problem = createOptimProblem('fminunc',...
    'objective',@(x)sawtoothxy(x(1),x(2)),...
    'x0',[100,-50],'options',...
    optimoptions(@fminunc,'Algorithm','quasi-newton','Display','off'));
```

3 Validate the problem structure by running it:

```
[x fval] = fminunc(problem)

x =
    8.4420 -110.2602

fval =
  435.2573
```

4 Create a default `MultiStart` object:

```
ms = MultiStart;
```

5 Run the solver for 50 iterations, recording the local minima:

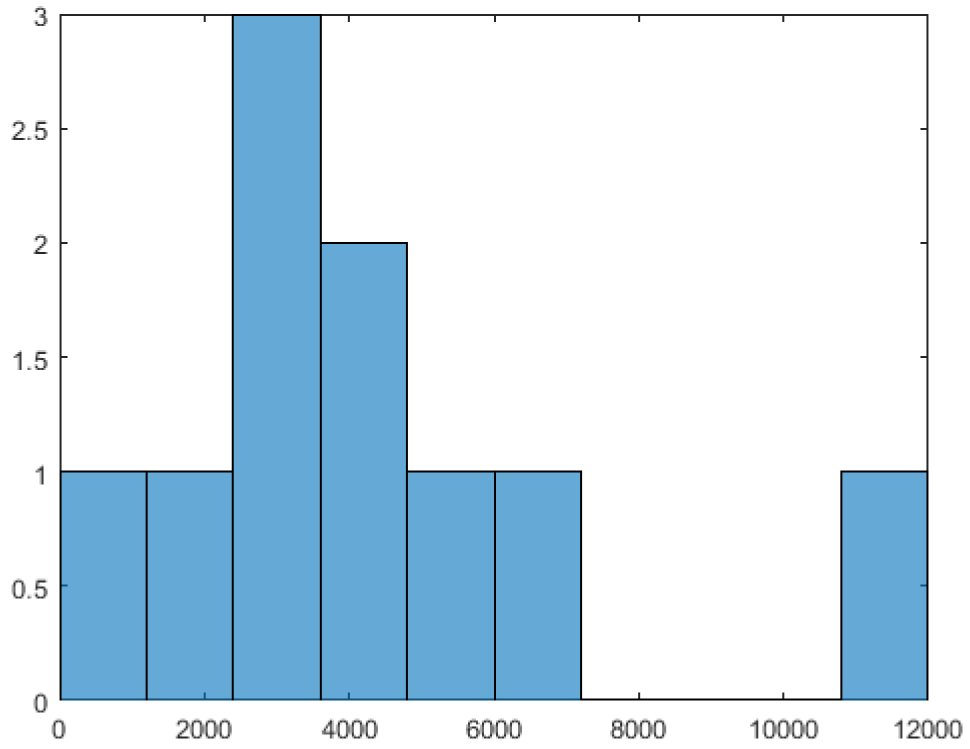
```
% rng(1) % uncomment to obtain the same result
[x fval eflag output manymins] = run(ms,problem,50)

MultiStart completed some of the runs from the start points.

10 out of 50 local solver runs converged with a positive local solver exit flag.

x =
  -73.8348 -197.7810

fval =
  766.8260

eflag =
     2

output =
                funcCount: 8583
         localSolverTotal: 50
       localSolverSuccess: 10
    localSolverIncomplete: 40
    localSolverNoSolution: 0
                  message: 'MultiStart completed some of the runs from the start po

manymins =
  1x10 GlobalOptimSolution array with properties:
    X
    Fval
    Exitflag
    Output
    X0
```

You can get different results, since `MultiStart` is stochastic.

The solver did not find the global minimum near `[0,0]`. It found 10 distinct local minima.

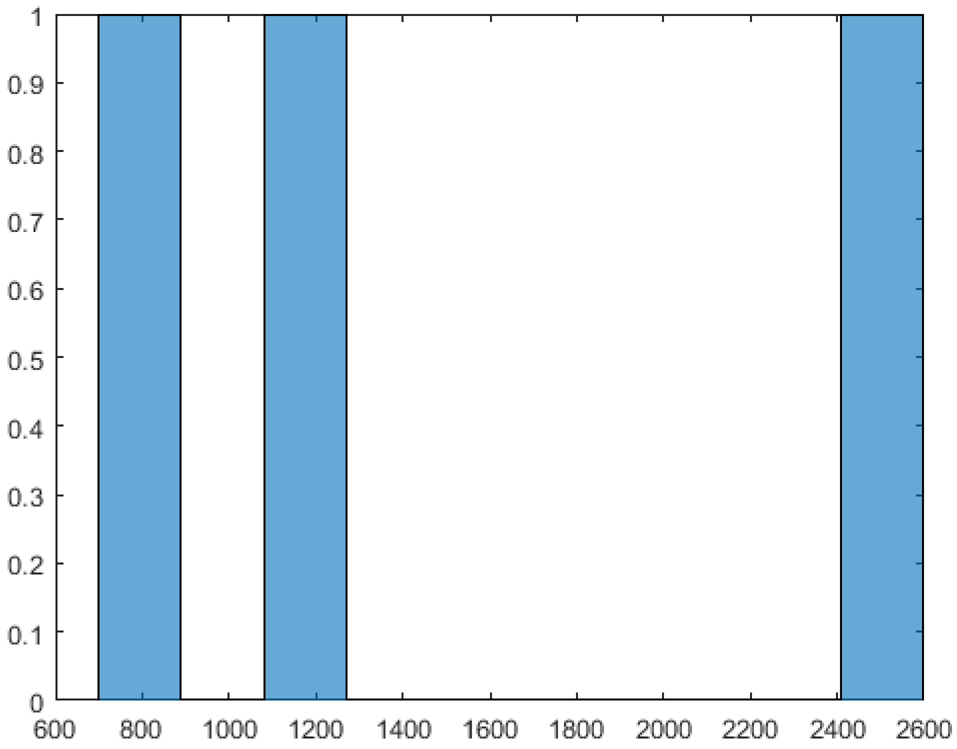
6 Plot the function values at the local minima:

```
histogram([manymins.Fval],10)
```



Plot the function values at the three best points:

```
bestf = [manymins.Fval];
histogram(bestf(1:3),10)
```



`MultiStart` started `fminunc` from start points with components uniformly distributed between –1000 and 1000. `fminunc` often got stuck in one of the many local minima. `fminunc` exceeded its iteration limit or function evaluation limit 40 times.

Optimize Using Only Feasible Start Points

You can set the `StartPointsToRun` option so that `MultiStart` and `GlobalSearch` use only start points that satisfy inequality constraints. This option can speed your optimization, since the local solver does not have to search for a feasible region. However, the option can cause the solvers to miss some basins of attraction.

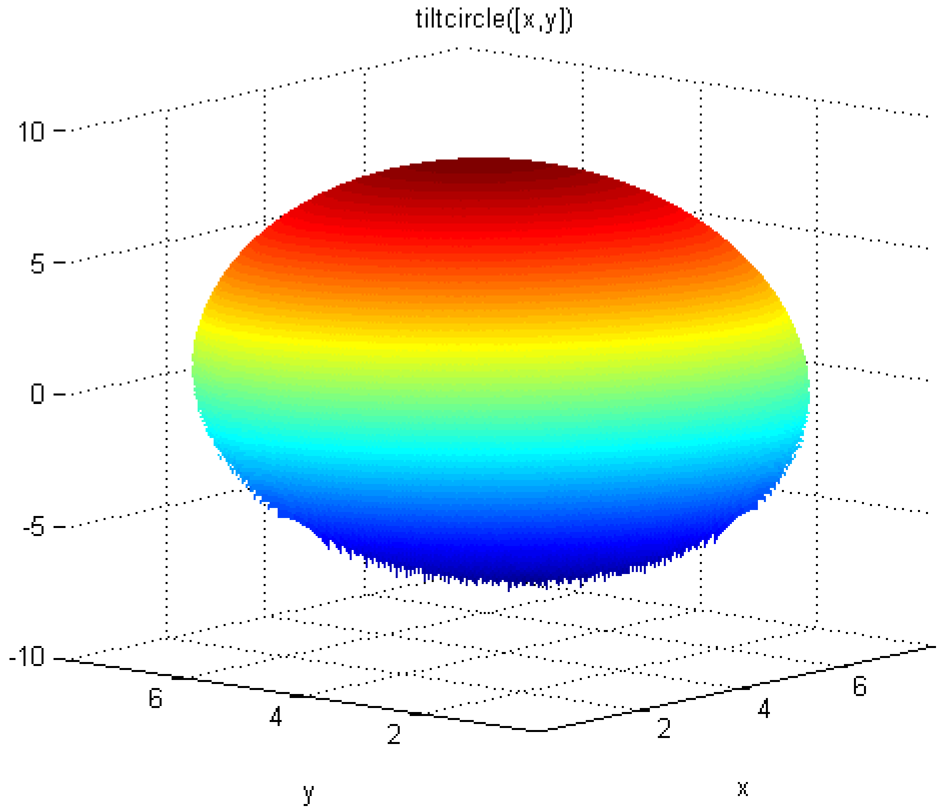
There are three settings for the `StartPointsToRun` option:

- `all` — Accepts all start points
- `bounds` — Rejects start points that do not satisfy bounds
- `bounds-ineqs` — Rejects start points that do not satisfy bounds or inequality constraints

For example, suppose your objective function is

```
function y = tiltcircle(x)
vx = x(:)-[4;4]; % ensure vx is in column form
y = vx'*[1;1] + sqrt(16 - vx'*vx); % complex if norm(x-[4;4])>4
```

`tiltcircle` returns complex values for `norm(x - [4 4]) > 4`.



Write a constraint function that is positive on the set where `norm(x - [4 4]) > 4`

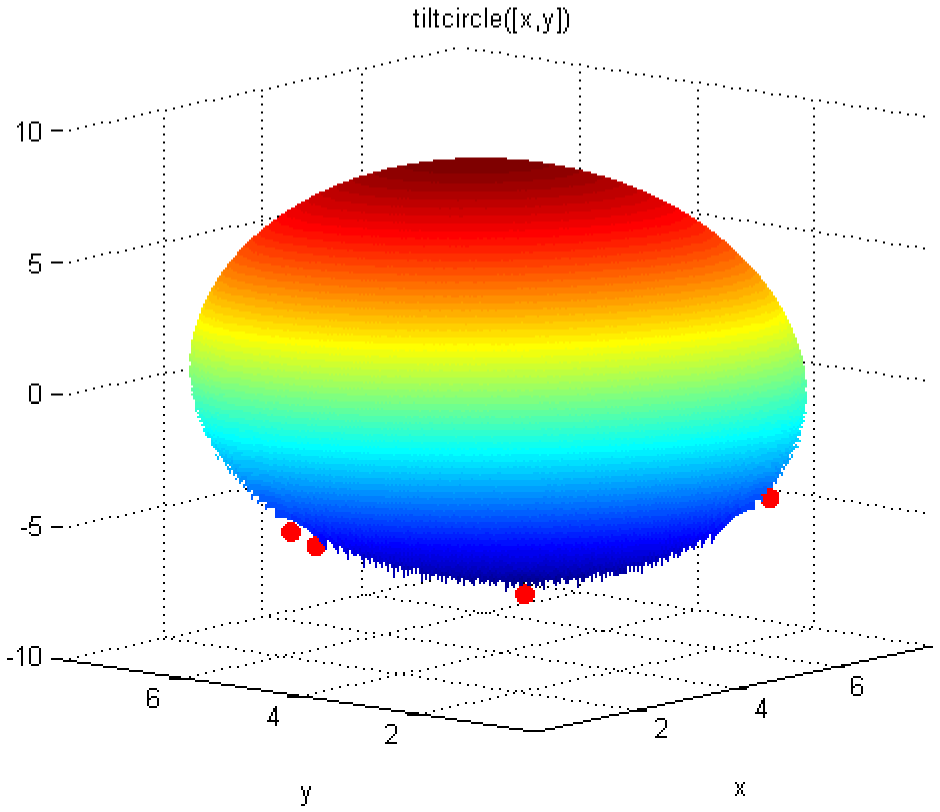
```
function [c ceq] = myconstraint(x)
ceq = [];
cx = x(:) - [4;4]; % ensure x is a column vector
c = cx'*cx - 16; % negative where tiltcircle(x) is real
```

Set `GlobalSearch` to use only start points satisfying inequality constraints:

```
gs = GlobalSearch('StartPointsToRun','bounds-ineqs');
```

To complete the example, create a problem structure and run the solver:

```
opts = optimoptions(@fmincon,'Algorithm','interior-point');
problem = createOptimProblem('fmincon',...
    'x0',[4 4],'objective',@tiltcircle,...
    'nonlcon',@myconstraint,'lb',[-10 -10],...
    'ub',[10 10],'options',opts);
rng(7,'twister'); % for reproducibility
[x,fval,exitflag,output,solutionset] = run(gs,problem)

GlobalSearch stopped because it analyzed all the trial points.

All 5 local solver runs converged with a positive local solver exit flag.

x =

    1.1716    1.1716


fval =

   -5.6530


exitflag =

     1


output =

                funcCount: 3256
         localSolverTotal: 5
       localSolverSuccess: 5
    localSolverIncomplete: 0
    localSolverNoSolution: 0
                  message: 'GlobalSearch stopped because it analyzed all the trial po

solutionset =

  1x4 GlobalOptimSolution array with properties:

    X
    Fval
    Exitflag
    Output
    X0
```



tiltcircle With Local Minima

The `tiltcircle` function has just one local minimum. Yet `GlobalSearch` (`fmincon`) stops at several points. Does this mean `fmincon` makes an error?

The reason that `fmincon` stops at several boundary points is subtle. The `tiltcircle` function has an infinite gradient on the boundary, as you can see from a one-dimensional calculation:

$$\frac{d}{dx}\sqrt{16-x^2} = \frac{-x}{\sqrt{16-x^2}} = \pm\infty \text{ at } |x| = 4.$$

So there is a huge gradient normal to the boundary. This gradient overwhelms the small additional tilt from the linear term. As far as `fmincon` can tell, boundary points are stationary points for the constrained problem.

This behavior can arise whenever you have a function that has a square root.

MultiStart Using lsqcurvefit or lsqnonlin

This example shows how to fit a function to data using `lsqcurvefit` together with `MultiStart`.

Many fitting problems have multiple local solutions. `MultiStart` can help find the global solution, meaning the best fit. While you can use `lsqnonlin` as the local solver, this example uses `lsqcurvefit` simply because it has a convenient syntax.

The model is

$$y = a + bx_1 sin(cx_2 + d),$$

where the input data is $x = (x_1, x_2)$, and the parameters a, b, c, and d are the unknown model coefficients.

Step 1. Create the objective function.

Write an anonymous function that takes a data matrix `xdata` with `N` rows and two columns, and returns a response vector with `N` rows. It also takes a coefficient matrix `p`, corresponding to the coefficient vector (a, b, c, d).

```
fitfcn = @(p,xdata)p(1) + p(2)*xdata(:,1).*sin(p(3)*xdata(:,2)+p(4));
```

Step 2. Create the training data.

Create 200 data points and responses. Use the values $a = -3, b = 1/4, c = 1/2, d = 1$. Include random noise in the response.

```
rng default % for reproducibility
N = 200; % number of data points
preal = [-3,1/4,1/2,1]; % real coefficients

xdata = 5*rand(N,2); % data points
ydata = fitfcn(preal,xdata) + 0.1*randn(N,1); % response data with noise
```

Step 3. Set bounds and initial point.

Set bounds for `lsqcurvefit`. There is no reason for d to exceed π in absolute value, because the sine function takes values in its full range over any interval of width

2π. Assume that the coefficient c must be smaller than 20 in absolute value, because allowing a very high frequency can cause unstable responses or spurious convergence.

```
lb = [-Inf,-Inf,-20,-pi];
ub = [Inf,Inf,20,pi];
```

Set the initial point arbitrarily to (5,5,5,0).

```
p0 = 5*ones(1,4); % Arbitrary initial point
p0(4) = 0; % so the initial point satisfies the bounds
```

Step 4. Find the best local fit.

Fit the parameters to the data, starting at `p0`.

```
[xfitted,errorfitted] = lsqcurvefit(fitfcn,p0,xdata,ydata,lb,ub)


Local minimum possible.

lsqcurvefit stopped because the final change in the sum of squares relative to
its initial value is less than the default value of the function tolerance.




xfitted =

   -2.6149   -0.0238    6.0191   -1.6998


errorfitted =

   28.2524
```

`lsqcurvefit` found a local solution that is not particularly close to the model parameter values (-3,1/4,1/2,1).

Step 5. Set up the problem for MultiStart.

Create a problem structure so `MultiStart` can solve the same problem.

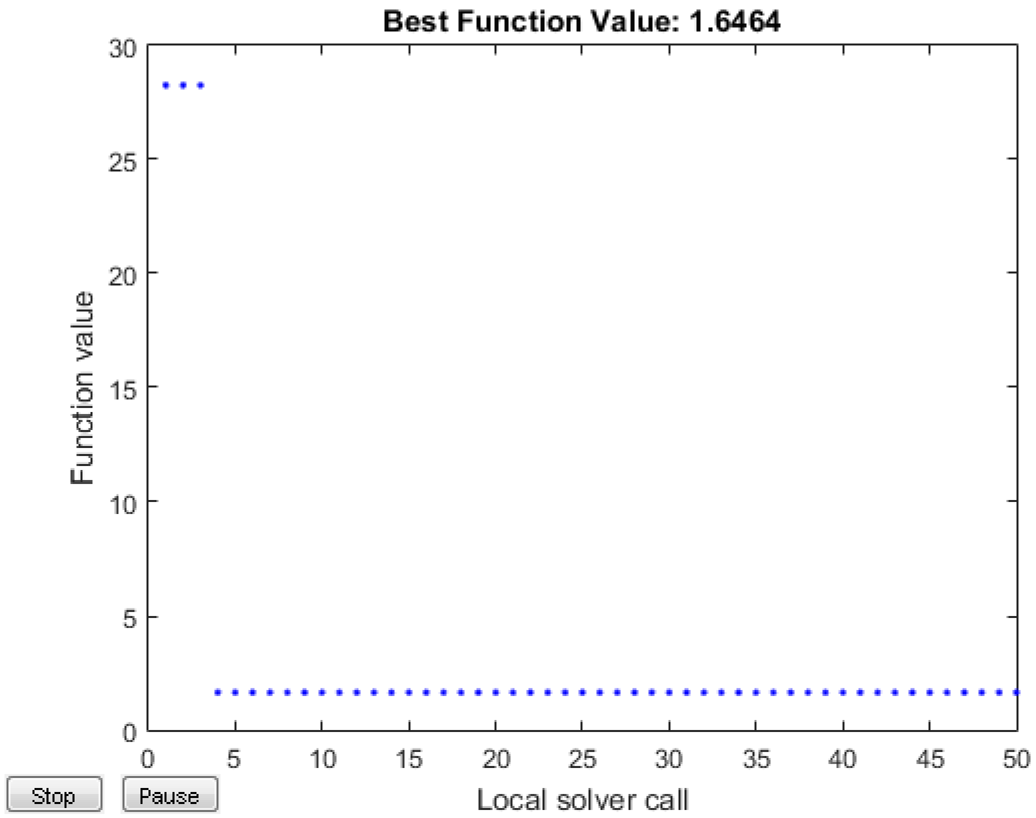
```
problem = createOptimProblem('lsqcurvefit','x0',p0,'objective',fitfcn,...
    'lb',lb,'ub',ub,'xdata',xdata,'ydata',ydata);
```

Step 6. Find a global solution.

Solve the fitting problem using `MultiStart` with 50 iterations. Plot the smallest error as the number of `MultiStart` iterations.

```
ms = MultiStart('PlotFcns',@gsplotbestf);
[xmulti,errormulti] = run(ms,problem,50)
```

```
MultiStart completed the runs from all start points.

All 50 local solver runs converged with a positive local solver exit flag.

xmulti =

   -2.9852   -0.2472   -0.4968   -1.0438


errormulti =

    1.6464
```



MultiStart found a global solution near the parameter values $(-3, -1/4, -1/2, -1)$. (This is equivalent to a solution near preal = $(-3, 1/4, 1/2, 1)$, because changing the sign of all the coefficients except the first gives the same numerical values of fitfcn.) The norm of the residual error decreased from about 28 to about 1.6, a decrease of more than a factor of 10.

Parallel MultiStart

In this section...
"Steps for Parallel MultiStart" on page 3-89
"Speedup with Parallel Computing" on page 3-91

Steps for Parallel MultiStart

If you have a multicore processor or access to a processor network, you can use Parallel Computing Toolbox™ functions with `MultiStart`. This example shows how to find multiple minima in parallel for a problem, using a processor with two cores. The problem is the same as in "Multiple Local Minima Via MultiStart" on page 3-76.

1 Write a function file to compute the objective:

```
function f = sawtoothxy(x,y)
[t r] = cart2pol(x,y); % change to polar coordinates
h = cos(2*t - 1/2)/2 + cos(t) + 2;
g = (sin(r) - sin(2*r)/2 + sin(3*r)/3 - sin(4*r)/4 + 4) ...
    .*r.^2./(r+1);
f = g.*h;
end
```

2 Create the problem structure:

```
problem = createOptimProblem('fminunc',...
    'objective',@(x)sawtoothxy(x(1),x(2)),...
    'x0',[100,-50],'options',...
    optimoptions(@fminunc,'Algorithm','quasi-newton'));
```

3 Validate the problem structure by running it:

```
[x fval] = fminunc(problem)

x =
    8.4420 -110.2602

fval =
  435.2573
```

4 Create a `MultiStart` object, and set the object to use parallel processing and iterative display:

```
ms = MultiStart('UseParallel',true,'Display','iter');
```

5 Set up parallel processing:

```
parpool

Starting parpool using the 'local' profile ... connected to 4 workers.

ans =

 Pool with properties:

            Connected: true
           NumWorkers: 4
              Cluster: local
        AttachedFiles: {}
          IdleTimeout: 30 minute(s) (30 minutes remaining)
          SpmdEnabled: true
```

6 Run the problem on 50 start points:

```
[x fval eflag output manymins] = run(ms,problem,50);
Running the local solvers in parallel.

 Run       Local       Local      Local    Local   First-order
Index     exitflag      f(x)     # iter   F-count   optimality
   17         2         3953         4       21        0.1626
   16         0         1331        45      201         65.02
   34         0         7271        54      201         520.9
   33         2         8249         4       18         2.968
     ... Many iterations omitted ...
   47         2         2740         5       21        0.0422
   35         0         8501        48      201         424.8
   50         0         1225        40      201         21.89

MultiStart completed some of the runs from the start points.

17 out of 50 local solver runs converged with a positive
local solver exit flag.
```

Notice that the run indexes look random. Parallel `MultiStart` runs its start points in an unpredictable order.

Notice that `MultiStart` confirms parallel processing in the first line of output, which states: “Running the local solvers in parallel.”

7 When finished, shut down the parallel environment:

```
delete(gcp)
```

```
    Parallel pool using the 'local' profile is shutting down.
```

For an example of how to obtain better solutions to this problem, see “Example: Searching for a Better Solution” on page 3-60. You can use parallel processing along with the techniques described in that example.

Speedup with Parallel Computing

The results of `MultiStart` runs are stochastic. The timing of runs is stochastic, too. Nevertheless, some clear trends are apparent in the following table. The data for the table came from one run at each number of start points, on a machine with two cores.

Start Points	Parallel Seconds	Serial Seconds
50	3.6	3.4
100	4.9	5.7
200	8.3	10
500	16	23
1000	31	46

Parallel computing can be slower than serial when you use only a few start points. As the number of start points increases, parallel computing becomes increasingly more efficient than serial.

There are many factors that affect speedup (or slowdown) with parallel processing. For more information, see “Improving Performance with Parallel Computing” in the Optimization Toolbox documentation.

Isolated Global Minimum

In this section...

Difficult-To-Locate Global Minimum

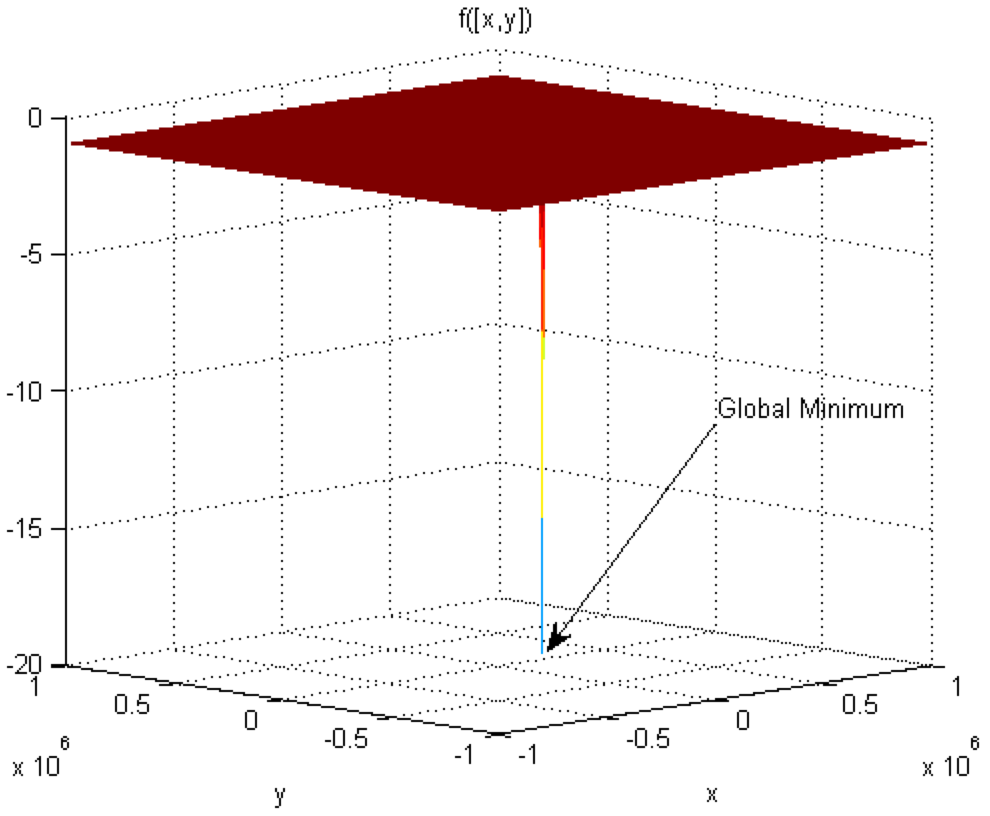
Finding a start point in the basin of attraction of the global minimum can be difficult when the basin is small or when you are unsure of the location of the minimum. To solve this type of problem you can:

- Add sensible bounds
- Take a huge number of random start points
- Make a methodical grid of start points
- For an unconstrained problem, take widely dispersed random start points

This example shows these methods and some variants.

The function –sech(x) is nearly 0 for all $|x| > 5$, and –sech(0) = –1. The example is a two-dimensional version of the sech function, with one minimum at `[1,1]`, the other at `[1e5,-1e5]`:
$f(x,y) = -10\text{sech}(|x-(1,1)|) - 20\text{sech}(.0003(|x-(1e5,-1e5)|) - 1.$

f has a global minimum of –21 at (1e5,–1e5), and a local minimum of –11 at (1,1).



The minimum at (1e5,–1e5) shows as a narrow spike. The minimum at (1,1) does not show since it is too narrow.

The following sections show various methods of searching for the global minimum. Some of the methods are not successful on this problem. Nevertheless, you might find each method useful for different problems.

Default Settings Cannot Find the Global Minimum — Add Bounds

`GlobalSearch` and `MultiStart` cannot find the global minimum using default global options, since the default start point components are in the range (–9999,10001) for `GlobalSearch` and (–1000,1000) for `MultiStart`.

With additional bounds of –1e6 and 1e6 in `problem`, `GlobalSearch` usually does not find the global minimum:

```
x1 = [1;1];x2 = [1e5;-1e5];
f = @(x)-10*sech(norm(x(:)-x1)) -20*sech((norm(x(:)-x2))*3e-4) -1;
opts = optimoptions(@fmincon,'Algorithm','active-set');
problem = createOptimProblem('fmincon','x0',[0,0],'objective',f,...
    'lb',[-1e6;-1e6],'ub',[1e6;1e6],'options',opts);
gs = GlobalSearch;
rng(14,'twister') % for reproducibility
[xfinal fval] = run(gs,problem)

GlobalSearch stopped because it analyzed all the trial points.

All 32 local solver runs converged with a positive
local solver exit flag.

xfinal =
    1.0000    1.0000

fval =
  -11.0000
```

GlobalSearch with Bounds and More Start Points

To find the global minimum, you can search more points. This example uses 1e5 start points, and a `MaxTime` of 300 s:

```
gs.NumTrialPoints = 1e5;
gs.MaxTime = 300;
[xg fvalg] = run(gs,problem)

GlobalSearch stopped because maximum time is exceeded.

GlobalSearch called the local solver 2186 times before exceeding
the clock time limit (MaxTime = 300 seconds).
1943 local solver runs converged with a positive
local solver exit flag.

xg =
   1.0e+04 *
   10.0000  -10.0000

fvalg =
  -21.0000
```

In this case, `GlobalSearch` found the global minimum.

MultiStart with Bounds and Many Start Points

Alternatively, you can search using `MultiStart` with many start points. This example uses 1e5 start points, and a `MaxTime` of 300 s:

```
ms = MultiStart(gs);
[xm fvalm] = run(ms,problem,1e5)

MultiStart stopped because maximum time was exceeded.

MultiStart called the local solver 17266 times before exceeding
the clock time limit (MaxTime = 300 seconds).
17266 local solver runs converged with a positive
local solver exit flag.

xm =
    1.0000    1.0000

fvalm =
  -11.0000
```

In this case, `MultiStart` failed to find the global minimum.

MultiStart Without Bounds, Widely Dispersed Start Points

You can also use `MultiStart` to search an unbounded region to find the global minimum. Again, you need many start points to have a good chance of finding the global minimum.

The first five lines of code generate 10,000 widely dispersed random start points using the method described in "Widely Dispersed Points for Unconstrained Components" on page 3-59. `newprob` is a problem structure using the `fminunc` local solver and no bounds:

```
rng(0,'twister') % for reproducibility
u = rand(1e4,1);
u = 1./u;
u = exp(u) - exp(1);
```

```
s = rand(1e4,1)*2*pi;
stpts = [u.*cos(s),u.*sin(s)];
startpts = CustomStartPointSet(stpts);

opts = optimoptions(@fminunc,'Algorithm','quasi-newton');
newprob = createOptimProblem('fminunc','x0',[0;0],'objective',f,...
    'options',opts);
[xcust fcust] = run(ms,newprob,startpts)

MultiStart completed the runs from all start points.

All 10000 local solver runs converged with a positive
local solver exit flag.

xcust =
   1.0e+05 *

    1.0000
   -1.0000

fcust =
  -21.0000
```

In this case, `MultiStart` found the global minimum.

MultiStart with a Regular Grid of Start Points

You can also use a grid of start points instead of random start points. To learn how to construct a regular grid for more dimensions, or one that has small perturbations, see "Uniform Grid" on page 3-58 or "Perturbed Grid" on page 3-59.

```
xx = -1e6:1e4:1e6;
[xxx yyy] = meshgrid(xx,xx);
z = [xxx(:),yyy(:)];
bigstart = CustomStartPointSet(z);
[xgrid fgrid] = run(ms,newprob,bigstart)

MultiStart completed the runs from all start points.

All 10000 local solver runs converged with a positive
local solver exit flag.

xgrid =
  1.0e+004 *
```

```
   10.0000
  -10.0000

fgrid =
  -21.0000
```

In this case, `MultiStart` found the global minimum.

MultiStart with Regular Grid and Promising Start Points

Making a regular grid of start points, especially in high dimensions, can use an inordinate amount of memory or time. You can filter the start points to run only those with small objective function value.

To perform this filtering most efficiently, write your objective function in a vectorized fashion. For information, see “Write a Vectorized Function” on page 2-3 or “Vectorize the Objective and Constraint Functions” on page 4-80. The following function handle computes a vector of objectives based on an input matrix whose rows represent start points:

```
x1 = [1;1];x2 = [1e5;-1e5];
g = @(x) -10*sech(sqrt((x(:,1)-x1(1)).^2 + (x(:,2)-x1(2)).^2)) ...
     -20*sech(sqrt((x(:,1)-x2(1)).^2 + (x(:,2)-x2(2)).^2))-1;
```

Suppose you want to run the local solver only for points where the value is less than –2. Start with a denser grid than in “MultiStart with a Regular Grid of Start Points” on page 3-96, then filter out all the points with high function value:

```
xx = -1e6:1e3:1e6;
[xxx yyy] = meshgrid(xx,xx);
z = [xxx(:),yyy(:)];
idx = g(z) < -2; % index of promising start points
zz = z(idx,:);
smallstartset = CustomStartPointSet(zz);
opts = optimoptions(@fminunc,'Algorithm','quasi-newton','Display','off');
newprobg = createOptimProblem('fminunc','x0',[0,0],...
    'objective',g,'options',opts);
    % row vector x0 since g expects rows
[xfew ffew] = run(ms,newprobg,smallstartset)

MultiStart completed the runs from all start points.
```

```
All 2 local solver runs converged with a positive
local solver exit flag.

xfew =
      100000     -100000

ffew =
   -21
```

In this case, `MultiStart` found the global minimum. There are only two start points in `smallstartset`, one of which is the global minimum.

Using Direct Search

What Is Direct Search?

Direct search is a method for solving optimization problems that does not require any information about the gradient of the objective function. Unlike more traditional optimization methods that use information about the gradient or higher derivatives to search for an optimal point, a direct search algorithm searches a set of points around the current point, looking for one where the value of the objective function is lower than the value at the current point. You can use direct search to solve problems for which the objective function is not differentiable, or is not even continuous.

Global Optimization Toolbox functions include three direct search algorithms called the generalized pattern search (GPS) algorithm, the generating set search (GSS) algorithm, and the mesh adaptive search (MADS) algorithm. All are *pattern search* algorithms that compute a sequence of points that approach an optimal point. At each step, the algorithm searches a set of points, called a *mesh*, around the *current point*—the point computed at the previous step of the algorithm. The mesh is formed by adding the current point to a scalar multiple of a set of vectors called a *pattern*. If the pattern search algorithm finds a point in the mesh that improves the objective function at the current point, the new point becomes the current point at the next step of the algorithm.

The GPS algorithm uses fixed direction vectors. The GSS algorithm is identical to the GPS algorithm, except when there are linear constraints, and when the current point is near a linear constraint boundary. The MADS algorithm uses a random selection of vectors to define the mesh. For details, see “Patterns” on page 4-12.

Optimize Using Pattern Search

In this section...

Call patternsearch at the Command Line

To perform a pattern search on an unconstrained problem at the command line, call the function `patternsearch` with the syntax

```
[x fval] = patternsearch(@objfun, x0)
```

where

- `@objfun` is a handle to the objective function.
- `x0` is the starting point for the pattern search.

The results are:

- `x` — Point at which the final value is attained
- `fval` — Final value of the objective function

Pattern Search on Unconstrained Problems

For an unconstrained problem, call `patternsearch` with the syntax

```
[x fval] = patternsearch(@objectfun, x0)
```

The output arguments are

- `x` — The final point
- `fval` — The value of the objective function at x

The required input arguments are

- `@objectfun` — A function handle to the objective function `objectfun`, which you can write as a function file. See “Compute Objective Functions” on page 2-2 to learn how to do this.
- `x0` — The initial point for the pattern search algorithm.

As an example, you can run the example described in “Optimize Using the GPS Algorithm” on page 4-8 from the command line by entering

```
[x fval] = patternsearch(@ps_example, [2.1 1.7])
```

This returns

```
Optimization terminated: mesh size less than options.TolMesh.

x =
   -4.7124   -0.0000

fval =
   -2.0000
```

Pattern Search on Constrained Problems

If your problem has constraints, use the syntax

```
[x fval] = patternsearch(@objfun,x0,A,b,Aeq,beq,lb,ub,nonlcon)
```

where

- `A` is a matrix and `b` is vector that represent inequality constraints of the form $A\cdot x \le b$.
- `Aeq` is a matrix and `beq` is a vector that represent equality constraints of the form $Aeq\cdot x = beq$.
- `lb` and `ub` are vectors representing bound constraints of the form $lb \le x$ and $x \le ub$, respectively.
- `nonlcon` is a function that returns the nonlinear equality and inequality vectors, c and ceq, respectively. The function is minimized such that $c(x) \le 0$ and $ceq(x) = 0$.

You only need to pass in the constraints that are part of the problem. For example, if there are no bound constraints or a nonlinear constraint function, use the syntax

```
[x fval] = patternsearch(@objfun,x0,A,b,Aeq,beq)
```

Use empty brackets `[]` for constraint arguments that are not needed for the problem. For example, if there are no inequality constraints or a nonlinear constraint function, use the syntax

```
[x fval] = patternsearch(@objfun,x0,[],[],Aeq,beq,lb,ub)
```

Additional Output Arguments

To get more information about the performance of the pattern search, you can call `patternsearch` with the syntax

```
[x fval exitflag output] = patternsearch(@objfun,x0)
```

Besides `x` and `fval`, this returns the following additional output arguments:

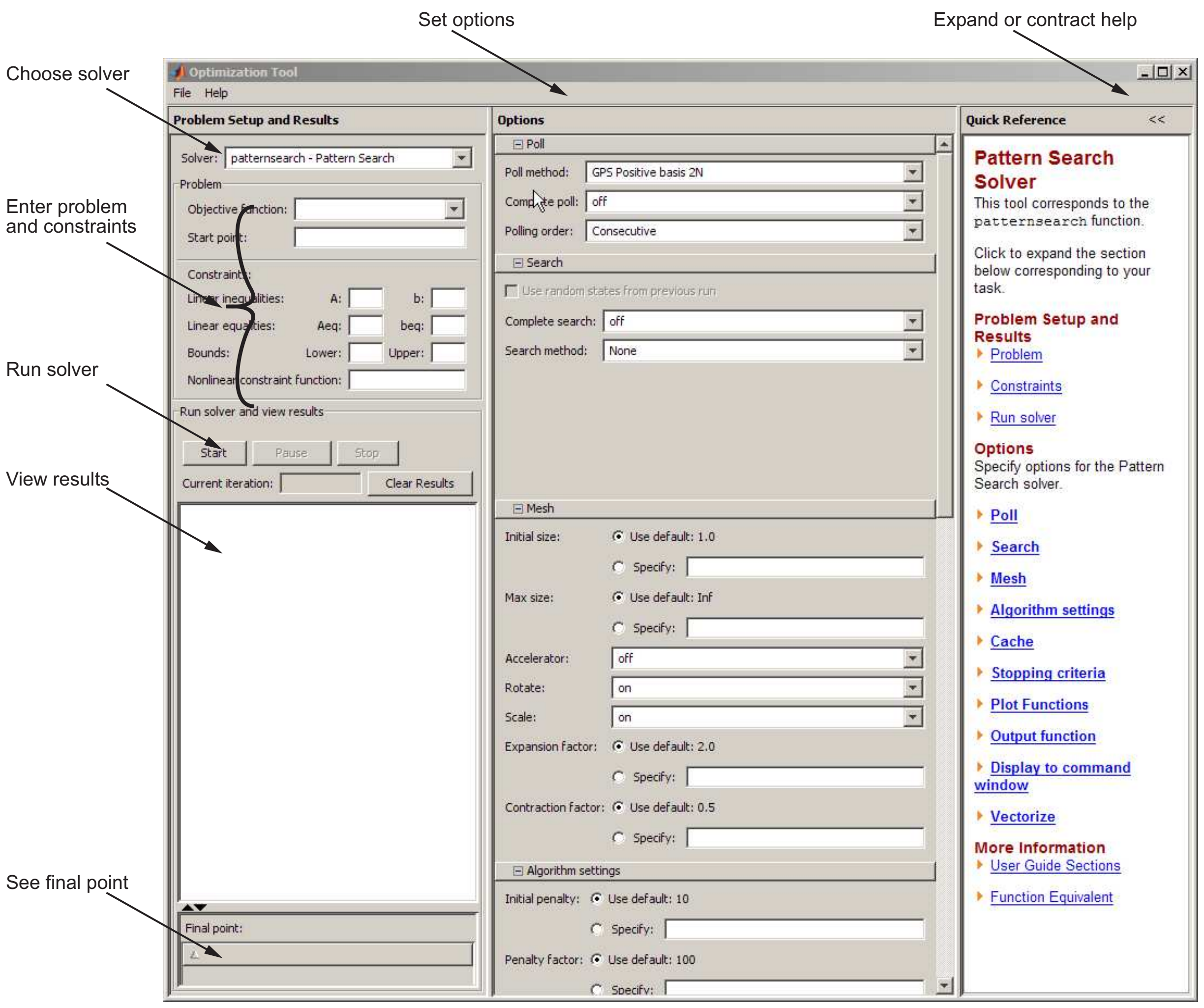
- `exitflag` — Integer indicating whether the algorithm was successful
- `output` — Structure containing information about the performance of the solver

For more information about these arguments, see the `patternsearch` reference page.

Use the Optimization App for Pattern Search

To open the Optimization app, enter

```
optimtool('patternsearch')
```
at the command line, or enter `optimtool` and then choose `patternsearch` from the **Solver** menu.



Problem Setup and Results

Solver: patternsearch - Pattern Search

Problem

Objective function:

Start point:

Constraints:

Linear inequalities: A: b:

Linear equalities: Aeq: beq:

Bounds: Lower: Upper:

Nonlinear constraint function:

Run solver and view results

Current iteration:

Final point:

Options

Poll

Poll method: GPS Positive basis 2N

Complete poll: off

Polling order: Consecutive

Search

Use random states from previous run

Complete search: off

Search method: None

Mesh

Initial size: Use default: 1.0 / Specify:

Max size: Use default: Inf / Specify:

Accelerator: off

Rotate: on

Scale: on

Expansion factor: Use default: 2.0 / Specify:

Contraction factor: Use default: 0.5 / Specify:

Algorithm settings

Initial penalty: Use default: 10 / Specify:

Penalty factor: Use default: 100 / Specify:

Quick Reference

Pattern Search Solver

This tool corresponds to the `patternsearch` function.

Click to expand the section below corresponding to your task.

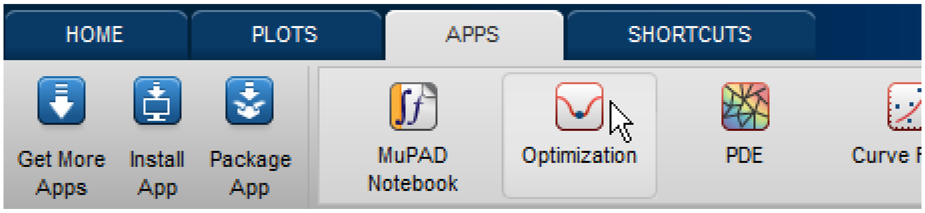
Problem Setup and Results
- Problem
- Constraints
- Run solver

Options

Specify options for the Pattern Search solver.
- Poll
- Search
- Mesh
- Algorithm settings
- Cache
- Stopping criteria
- Plot Functions
- Output function
- Display to command window
- Vectorize

More Information
- User Guide Sections
- Function Equivalent

You can also start the tool from the MATLAB **Apps** tab.



To use the Optimization app, first enter the following information:

- **Objective function** — The objective function you want to minimize. Enter the objective function in the form `@objfun`, where `objfun.m` is a file that computes the objective function. The `@` sign creates a function handle to `objfun`.
- **Start point** — The initial point at which the algorithm starts the optimization.

In the **Constraints** pane, enter linear constraints, bounds, or a nonlinear constraint function as a function handle for the problem. If the problem is unconstrained, leave these fields blank.

Then, click **Start**. The tool displays the results of the optimization in the **Run solver and view results** pane.

In the **Options** pane, set the options for the pattern search. To view the options in a category, click the + sign next to it.

"Finding the Minimum of the Function" on page 4-8 gives an example of using the Optimization app.

For more information, see "Optimization App" in the Optimization Toolbox documentation.

Optimize Using the GPS Algorithm

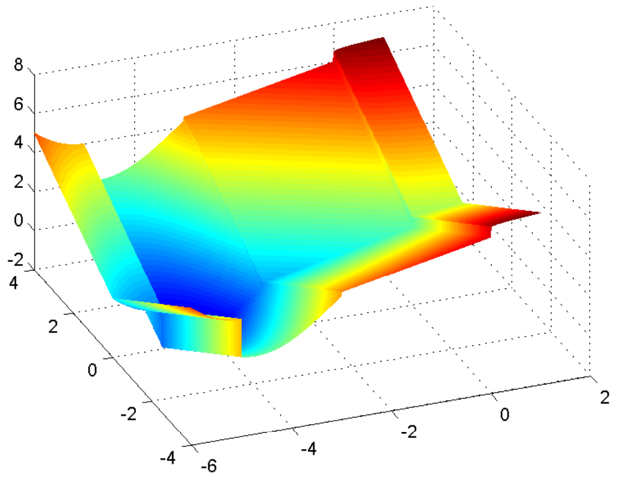
In this section...

Objective Function

This example uses the objective function, `ps_example`, which is included with Global Optimization Toolbox software. View the code for the function by entering

```
type ps_example
```

The following figure shows a plot of the function.



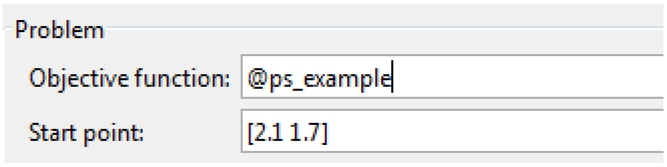
Finding the Minimum of the Function

To find the minimum of `ps_example`, perform the following steps:

1 Enter

   ```
   optimtool
   ```

   and then choose the `patternsearch` solver.
2 In the **Objective function** field of the Optimization app, enter `@ps_example`.
3 In the **Start point** field, type `[2.1 1.7]`.



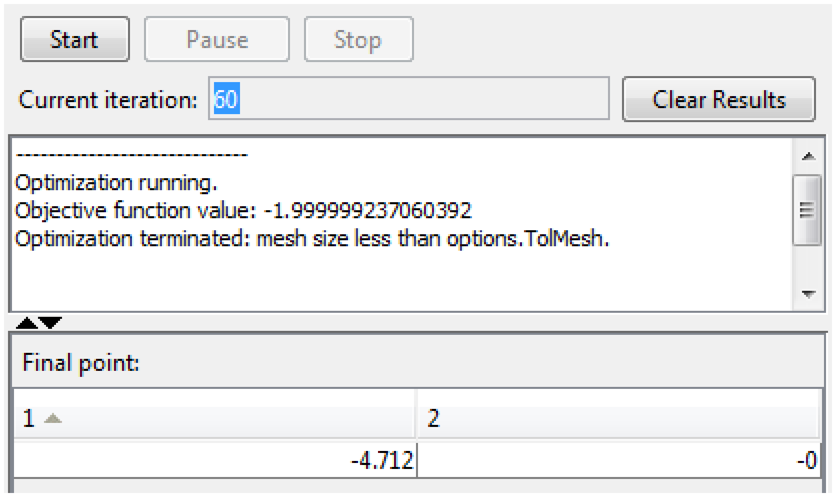
Problem

Objective function: @ps_example

Start point: [2.1 1.7]

   Leave the fields in the **Constraints** pane blank because the problem is unconstrained.
4 Click **Start** to run the pattern search.

The **Run solver and view results** pane displays the results of the pattern search.



Current iteration: 60

```
------------------------------
Optimization running.
Objective function value: -1.999999237060392
Optimization terminated: mesh size less than options.TolMesh.
```

Final point:

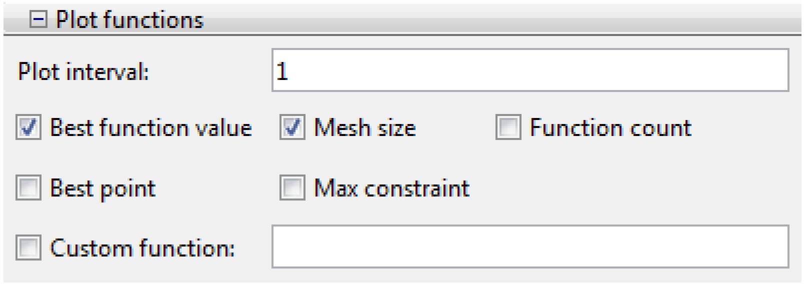
1	2
-4.712	-0

The reason the optimization terminated is that the mesh size became smaller than the acceptable tolerance value for the mesh size, defined by the **Mesh tolerance** parameter in the **Stopping criteria** pane. The minimum function value is approximately –2. The **Final point** pane displays the point at which the minimum occurs.

Plotting the Objective Function Values and Mesh Sizes

To see the performance of the pattern search, display plots of the best function value and mesh size at each iteration. First, select the following check boxes in the **Plot functions** pane:

- **Best function value**
- **Mesh size**



Plot functions

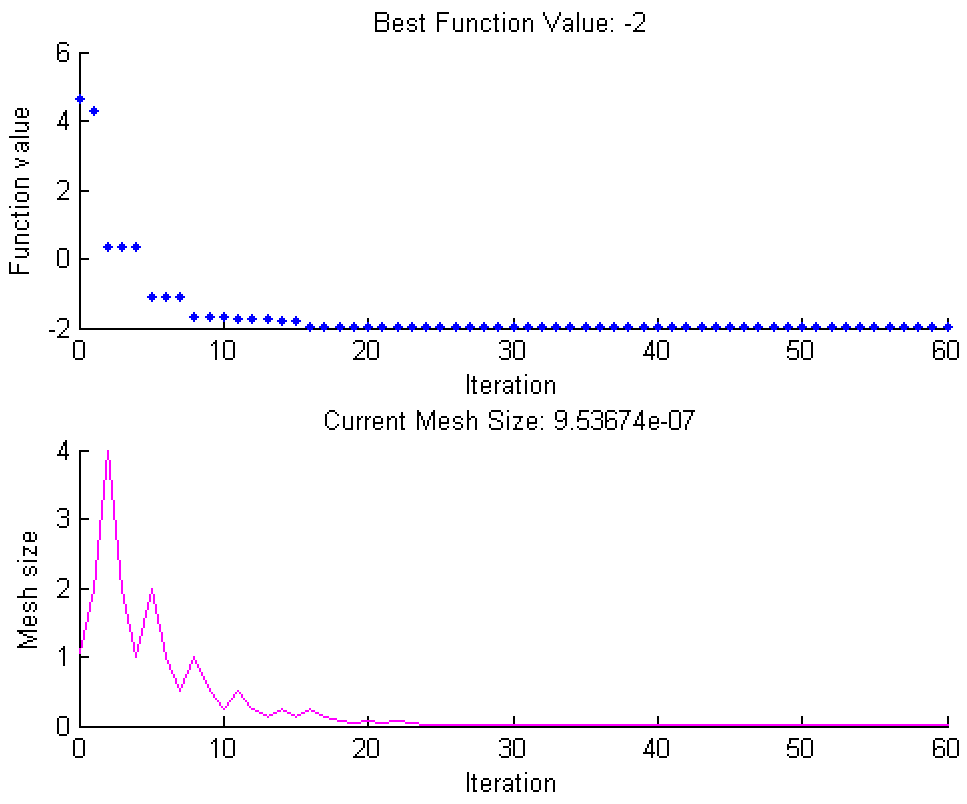
Plot interval: 1

[✓] Best function value [✓] Mesh size [ ] Function count

[ ] Best point [ ] Max constraint

[ ] Custom function:

Then click **Start** to run the pattern search. This displays the following plots.



The upper plot shows the objective function value of the best point at each iteration. Typically, the objective function values improve rapidly at the early iterations and then level off as they approach the optimal value.

The lower plot shows the mesh size at each iteration. The mesh size increases after each successful iteration and decreases after each unsuccessful one, explained in “How Pattern Search Polling Works” on page 4-15.

Pattern Search Terminology

In this section...

Patterns

A *pattern* is a set of vectors $\{v_i\}$ that the pattern search algorithm uses to determine which points to search at each iteration. The set $\{v_i\}$ is defined by the number of independent variables in the objective function, N, and the positive basis set. Two commonly used positive basis sets in pattern search algorithms are the maximal basis, with $2N$ vectors, and the minimal basis, with $N+1$ vectors.

With GPS, the collection of vectors that form the pattern are fixed-direction vectors. For example, if there are three independent variables in the optimization problem, the default for a $2N$ positive basis consists of the following pattern vectors:

$$\begin{aligned} v_1 &= [1 \quad 0 \quad 0] & v_2 &= [0 \quad 1 \quad 0] & v_3 &= [0 \quad 0 \quad 1] \\ v_4 &= [-1 \quad 0 \quad 0] & v_5 &= [0 \quad -1 \quad 0] & v_6 &= [0 \quad 0 \quad -1] \end{aligned}$$

An $N+1$ positive basis consists of the following default pattern vectors.

$$\begin{aligned} v_1 &= [1 \quad 0 \quad 0] & v_2 &= [0 \quad 1 \quad 0] & v_3 &= [0 \quad 0 \quad 1] \\ v_4 &= [-1 \quad -1 \quad -1] \end{aligned}$$

With GSS, the pattern is identical to the GPS pattern, except when there are linear constraints and the current point is near a constraint boundary. For a description of the way in which GSS forms a pattern with linear constraints, see Kolda, Lewis, and Torczon [1]. The GSS algorithm is more efficient than the GPS algorithm when you have linear constraints. For an example showing the efficiency gain, see "Compare the Efficiency of Poll Options" on page 4-51.

With MADS, the collection of vectors that form the pattern are randomly selected by the algorithm. Depending on the poll method choice, the number of vectors selected will be

$2N$ or N+1. As in GPS, $2N$ vectors consist of N vectors and their N negatives, while N+1 vectors consist of N vectors and one that is the negative of the sum of the others.

References

[1] Kolda, Tamara G., Robert Michael Lewis, and Virginia Torczon. "A generating set direct search augmented Lagrangian algorithm for optimization with a combination of general and linear constraints." Technical Report SAND2006-5315, Sandia National Laboratories, August 2006.

Meshes

At each step, `patternsearch` searches a set of points, called a *mesh*, for a point that improves the objective function. `patternsearch` forms the mesh by

1. Generating a set of vectors $\{d_i\}$ by multiplying each pattern vector v_i by a scalar Δ^m. Δ^m is called the *mesh size*.
2. Adding the $\{d_i\}$ to the *current point*—the point with the best objective function value found at the previous step.

For example, using the GPS algorithm. suppose that:

- The current point is `[1.6 3.4]`.
- The pattern consists of the vectors

$$\begin{aligned} v_1 &= [1 \quad 0] \\ v_2 &= [0 \quad 1] \\ v_3 &= [-1 \quad 0] \\ v_4 &= [0 \quad -1] \end{aligned}$$

- The current mesh size Δ^m is 4.

The algorithm multiplies the pattern vectors by 4 and adds them to the current point to obtain the following mesh.

```
[1.6 3.4] + 4*[1 0] = [5.6 3.4]
[1.6 3.4] + 4*[0 1] = [1.6 7.4]
[1.6 3.4] + 4*[-1 0] = [-2.4 3.4]
```

```
[1.6 3.4] + 4*[0 -1] = [1.6 -0.6]
```

The pattern vector that produces a mesh point is called its *direction*.

Polling

At each step, the algorithm polls the points in the current mesh by computing their objective function values. When the **Complete poll** option has the (default) setting `Off`, the algorithm stops polling the mesh points as soon as it finds a point whose objective function value is less than that of the current point. If this occurs, the poll is called *successful* and the point it finds becomes the current point at the next iteration.

The algorithm only computes the mesh points and their objective function values up to the point at which it stops the poll. If the algorithm fails to find a point that improves the objective function, the poll is called *unsuccessful* and the current point stays the same at the next iteration.

When the **Complete poll** option has the setting `On`, the algorithm computes the objective function values at all mesh points. The algorithm then compares the mesh point with the smallest objective function value to the current point. If that mesh point has a smaller value than the current point, the poll is successful.

Expanding and Contracting

After polling, the algorithm changes the value of the mesh size Δ^m. The default is to multiply Δ^m by 2 after a successful poll, and by 0.5 after an unsuccessful poll.

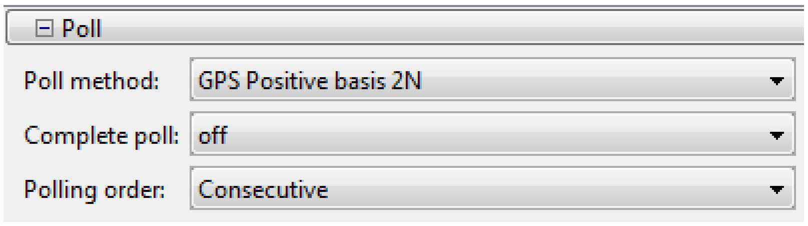
How Pattern Search Polling Works

In this section...

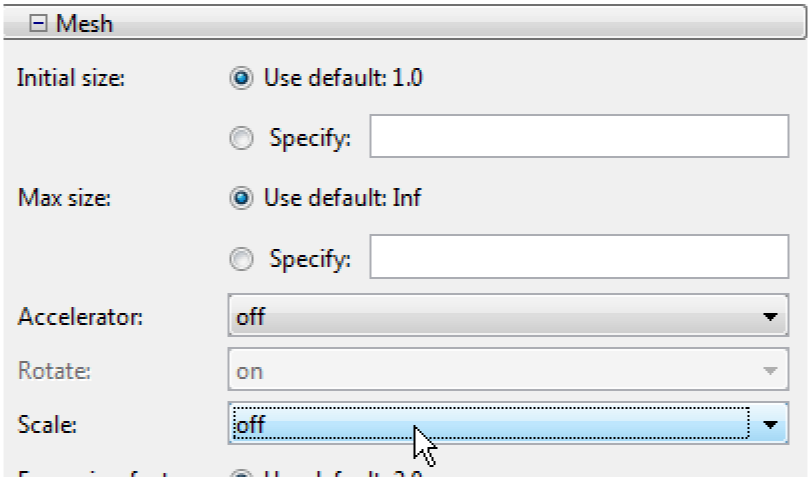
Context

`patternsearch` finds a sequence of points, `x0`, `x1`, `x2`, ... , that approach an optimal point. The value of the objective function either decreases or remains the same from each point in the sequence to the next. This section explains how pattern search works for the function described in "Optimize Using the GPS Algorithm" on page 4-8.

To simplify the explanation, this section describes how the generalized pattern search (GPS) works using a maximal positive basis of 2*N*, with **Scale** set to `Off` in **Mesh** options.



Poll method: GPS Positive basis 2N
Complete poll: off
Polling order: Consecutive



Initial size: (● Use default: 1.0) ○ Specify:

Max size: (● Use default: Inf) ○ Specify:

Accelerator: off

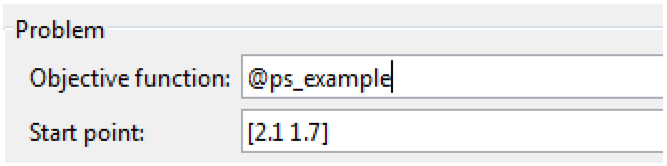
Rotate: on

Scale: off

This section does not show how the `patternsearch` algorithm works with bounds or linear constraints. For bounds and linear constraints, `patternsearch` modifies poll points to be feasible, meaning to satisfy all bounds and linear constraints.

This section does not encompass nonlinear constraints. To understand how `patternsearch` works with nonlinear constraints, see "Nonlinear Constraint Solver Algorithm" on page 4-37.

The problem setup:



Objective function: @ps_example

Start point: [2.1 1.7]

Successful Polls

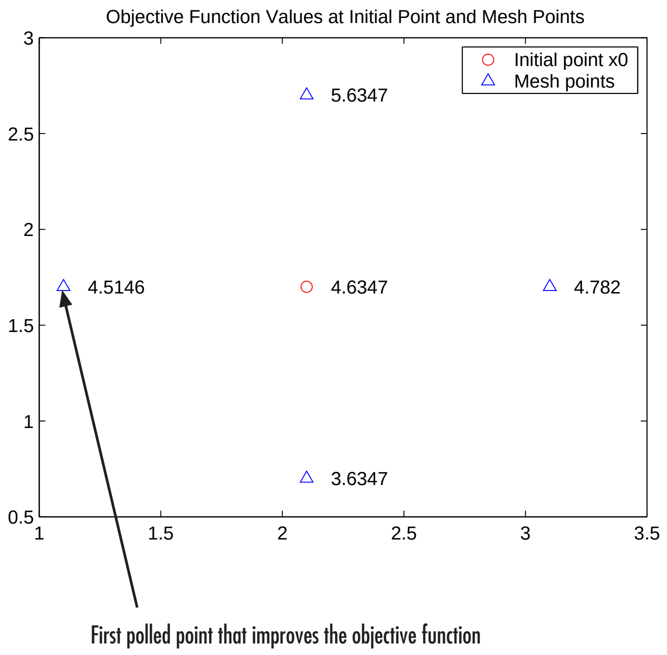
The pattern search begins at the initial point `x0` that you provide. In this example, `x0 = [2.1 1.7]`.

Iteration 1

At the first iteration, the mesh size is 1 and the GPS algorithm adds the pattern vectors to the initial point `x0 = [2.1 1.7]` to compute the following mesh points:

```
[1 0] + x0 = [3.1 1.7]
[0 1] + x0 = [2.1 2.7]
[-1 0] + x0 = [1.1 1.7]
[0 -1] + x0 = [2.1 0.7]
```

The algorithm computes the objective function at the mesh points in the order shown above. The following figure shows the value of `ps_example` at the initial point and mesh points.



The algorithm polls the mesh points by computing their objective function values until it finds one whose value is smaller than 4.6347, the value at `x0`. In this case, the first such point it finds is `[1.1 1.7]`, at which the value of the objective function is `4.5146`, so the poll at iteration 1 is *successful*. The algorithm sets the next point in the sequence equal to

```
x1 = [1.1 1.7]
```

Note By default, the GPS pattern search algorithm stops the current iteration as soon as it finds a mesh point whose fitness value is smaller than that of the current point.

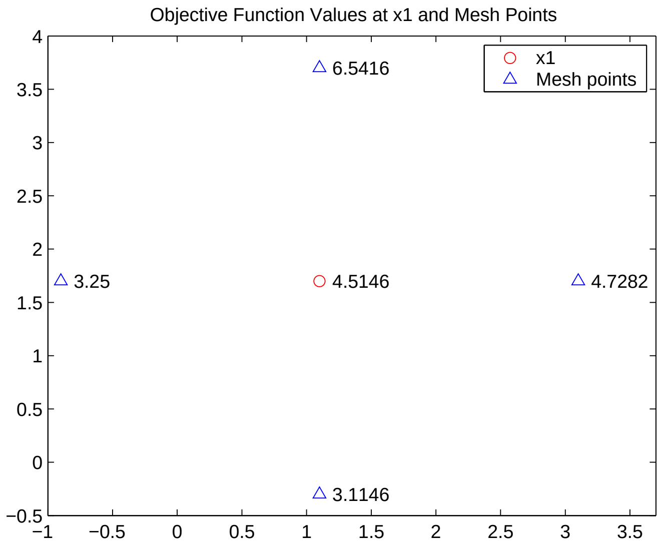
Consequently, the algorithm might not poll all the mesh points. You can make the algorithm poll all the mesh points by setting **Complete poll** to On.

Iteration 2

After a successful poll, the algorithm multiplies the current mesh size by 2, the default value of **Expansion factor** in the **Mesh** options pane. Because the initial mesh size is 1, at the second iteration the mesh size is 2. The mesh at iteration 2 contains the following points:

```
2*[1 0] + x1 = [3.1 1.7]
2*[0 1] + x1 = [1.1 3.7]
2*[-1 0] + x1 = [-0.9 1.7]
2*[0 -1] + x1 = [1.1 -0.3]
```

The following figure shows the point x1 and the mesh points, together with the corresponding values of ps_example.



The algorithm polls the mesh points until it finds one whose value is smaller than 4.5146, the value at x1. The first such point it finds is [-0.9 1.7], at which the value of the objective function is 3.25, so the poll at iteration 2 is again successful. The algorithm sets the second point in the sequence equal to

```
x2 = [-0.9 1.7]
```

Because the poll is successful, the algorithm multiplies the current mesh size by 2 to get a mesh size of 4 at the third iteration.

An Unsuccessful Poll

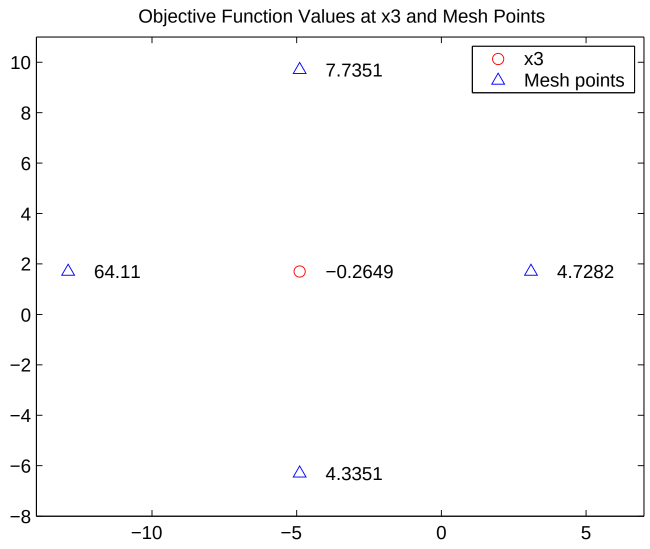
By the fourth iteration, the current point is

```
x3 = [-4.9 1.7]
```

and the mesh size is 8, so the mesh consists of the points

```
8*[1 0] + x3 = [3.1 1.7]
8*[0 1] + x3 = [-4.9 9.7]
8*[-1 0] + x3 = [-12.9 1.7]
8*[0 -1] + x3 = [-4.9 -1.3]
```

The following figure shows the mesh points and their objective function values.



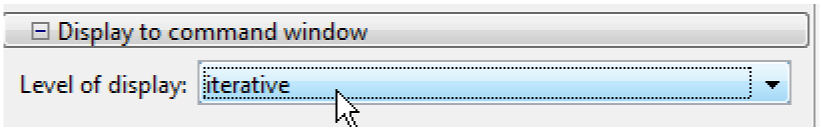
At this iteration, none of the mesh points has a smaller objective function value than the value at x3, so the poll is *unsuccessful*. In this case, the algorithm does not change the current point at the next iteration. That is,

```
x4 = x3;
```

At the next iteration, the algorithm multiplies the current mesh size by 0.5, the default value of **Contraction factor** in the **Mesh** options pane, so that the mesh size at the next iteration is 4. The algorithm then polls with a smaller mesh size.

Displaying the Results at Each Iteration

You can display the results of the pattern search at each iteration by setting **Level of display** to `Iterative` in the **Display to command window** options. This enables you to evaluate the progress of the pattern search and to make changes to `options` if necessary.



Level of display: iterative

With this setting, the pattern search displays information about each iteration at the command line. The first four iterations are

```
Iter     f-count          f(x)      MeshSize     Method
    0        1         4.63474          1
    1        4         4.51464          2     Successful Poll
    2        7            3.25          4     Successful Poll
    3       10       -0.264905          8     Successful Poll
    4       14       -0.264905          4     Refine Mesh
```

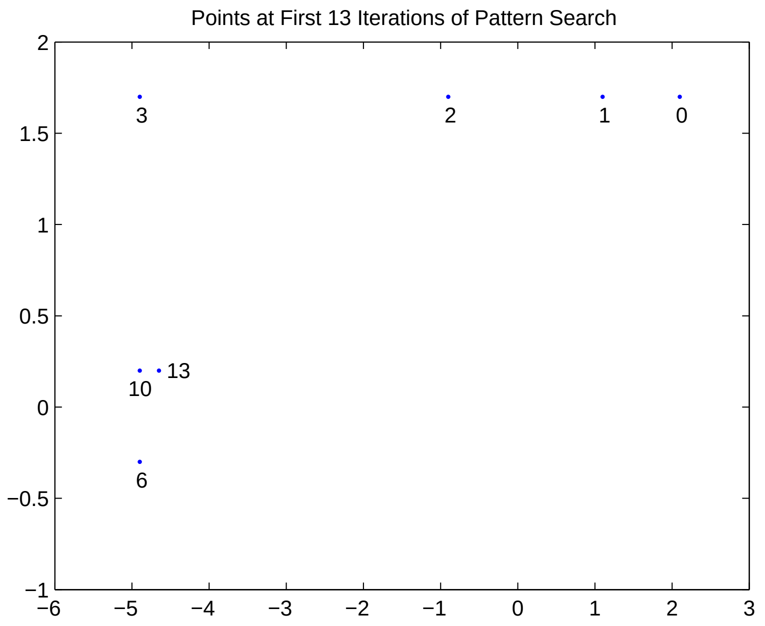
The entry `Successful Poll` below `Method` indicates that the current iteration was successful. For example, the poll at iteration 2 is successful. As a result, the objective function value of the point computed at iteration 2, displayed below `f(x)`, is less than the value at iteration 1.

At iteration 4, the entry `Refine Mesh` tells you that the poll is unsuccessful. As a result, the function value at iteration 4 remains unchanged from iteration 3.

By default, the pattern search doubles the mesh size after each successful poll and halves it after each unsuccessful poll.

More Iterations

The pattern search performs 60 iterations before stopping. The following plot shows the points in the sequence computed in the first 13 iterations of the pattern search.



The numbers below the points indicate the first iteration at which the algorithm finds the point. The plot only shows iteration numbers corresponding to successful polls, because the best point doesn't change after an unsuccessful poll. For example, the best point at iterations 4 and 5 is the same as at iteration 3.

Poll Method

At each iteration, the pattern search polls the points in the current mesh—that is, it computes the objective function at the mesh points to see if there is one whose function value is less than the function value at the current point. "How Pattern Search Polling Works" on page 4-15 provides an example of polling. You can specify the pattern that defines the mesh by the **Poll method** option. The default pattern, `GPS Positive basis 2N`, consists of the following $2N$ directions, where N is the number of independent variables for the objective function.
[1 0 0...0]
[0 1 0...0]
...
[0 0 0...1]

[–1 0 0...0]
[0 –1 0...0]
[0 0 0...–1].

For example, if the objective function has three independent variables, the `GPS Positive basis 2N`, consists of the following six vectors.
[1 0 0]
[0 1 0]
[0 0 1]
[–1 0 0]
[0 –1 0]
[0 0 –1].

Alternatively, you can set **Poll method** to `GPS Positive basis NP1`, the pattern consisting of the following $N + 1$ directions.
[1 0 0...0]
[0 1 0...0]
...
[0 0 0...1]
[–1 –1 –1...–1].

For example, if objective function has three independent variables, the `GPS Positive basis Np1`, consists of the following four vectors.
[1 0 0]
[0 1 0]
[0 0 1]
[–1 –1 –1].

A pattern search will sometimes run faster using `GPS Positive basis Np1` rather than the `GPS Positive basis 2N` as the **Poll method**, because the algorithm searches fewer points at each iteration. Although not being addressed in this example, the same is true when using the `MADS Positive basis Np1` over the `MADS Positive basis 2N`, and similarly for GSS. For example, if you run a pattern search on the example described in “Linearly Constrained Problem” on page 4-69, the algorithm performs 1588 function evaluations with `GPS Positive basis 2N`, the default **Poll method**, but only 877 function evaluations using `GPS Positive basis Np1`. For more detail, see “Compare the Efficiency of Poll Options” on page 4-51.

However, if the objective function has many local minima, using `GPS Positive basis 2N` as the **Poll method** might avoid finding a local minimum that is not the global minimum, because the search explores more points around the current point at each iteration.

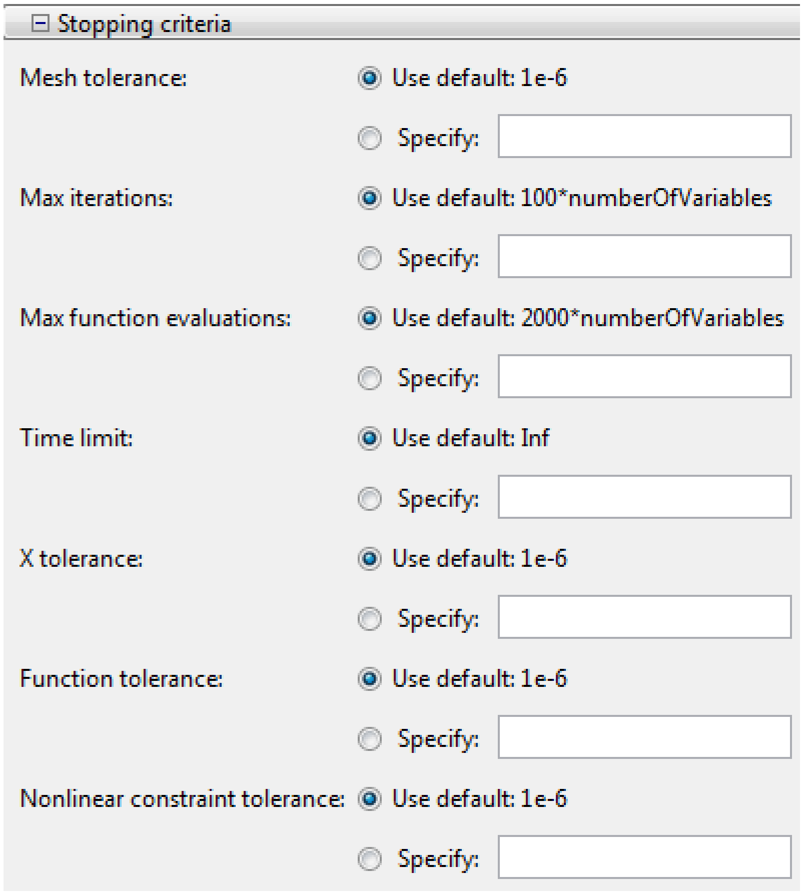
Complete Poll

By default, if the pattern search finds a mesh point that improves the value of the objective function, it stops the poll and sets that point as the current point for the next iteration. When this occurs, some mesh points might not get polled. Some of these unpolled points might have an objective function value that is even lower than the first one the pattern search finds.

For problems in which there are several local minima, it is sometimes preferable to make the pattern search poll *all* the mesh points at each iteration and choose the one with the best objective function value. This enables the pattern search to explore more points at each iteration and thereby potentially avoid a local minimum that is not the global minimum. In the Optimization app you can make the pattern search poll the entire mesh setting **Complete poll** to `On` in **Poll** options. At the command line, use `psoptimset` to set the `CompletePoll` option to `'on'`.

Stopping Conditions for the Pattern Search

The criteria for stopping the pattern search algorithm are listed in the **Stopping criteria** section of the Optimization app:


⊟ Stopping criteria

Mesh tolerance:	[✓ Use default: 1e-6] Specify:
Max iterations:	[✓ Use default: 100*numberOfVariables] Specify:
Max function evaluations:	[✓ Use default: 2000*numberOfVariables] Specify:
Time limit:	[✓ Use default: Inf] Specify:
X tolerance:	[✓ Use default: 1e-6] Specify:
Function tolerance:	[✓ Use default: 1e-6] Specify:
Nonlinear constraint tolerance:	[✓ Use default: 1e-6] Specify:

The algorithm stops when any of the following conditions occurs:

- The mesh size is less than **Mesh tolerance**.
- The number of iterations performed by the algorithm reaches the value of **Max iteration**.
- The total number of objective function evaluations performed by the algorithm reaches the value of **Max function evaluations**.
- The time, in seconds, the algorithm runs until it reaches the value of **Time limit**.
- The distance between the point found in two consecutive iterations and the mesh size are both less than **X tolerance**.

- The change in the objective function in two consecutive iterations and the mesh size are both less than **Function tolerance**.

Nonlinear constraint tolerance is not used as stopping criterion. It determines the feasibility with respect to nonlinear constraints.

The MADS algorithm uses an additional parameter called the poll parameter, Δ_p, in the mesh size stopping criterion:

$$\Delta_p = \begin{cases} N\sqrt{\Delta_m} & \text{for positive basis } N+1 \text{ poll} \\ \sqrt{\Delta_m} & \text{for positive basis } 2N \text{ poll,} \end{cases}$$

where Δ_m is the mesh size. The MADS stopping criterion is:
$\Delta_p \leq$ **Mesh tolerance**.

Robustness of Pattern Search

The pattern search algorithm is robust in relation to objective function failures. This means `patternsearch` tolerates function evaluations resulting in `NaN`, `Inf`, or complex values. When the objective function at the initial point `x0` is a real, finite value, `patternsearch` treats poll point failures as if the objective function values are large, and ignores them.

For example, if all points in a poll evaluate to `NaN`, `patternsearch` considers the poll unsuccessful, shrinks the mesh, and reevaluates. If even one point in a poll evaluates to a smaller value than any seen yet, `patternsearch` considers the poll successful, and expands the mesh.

Searching and Polling

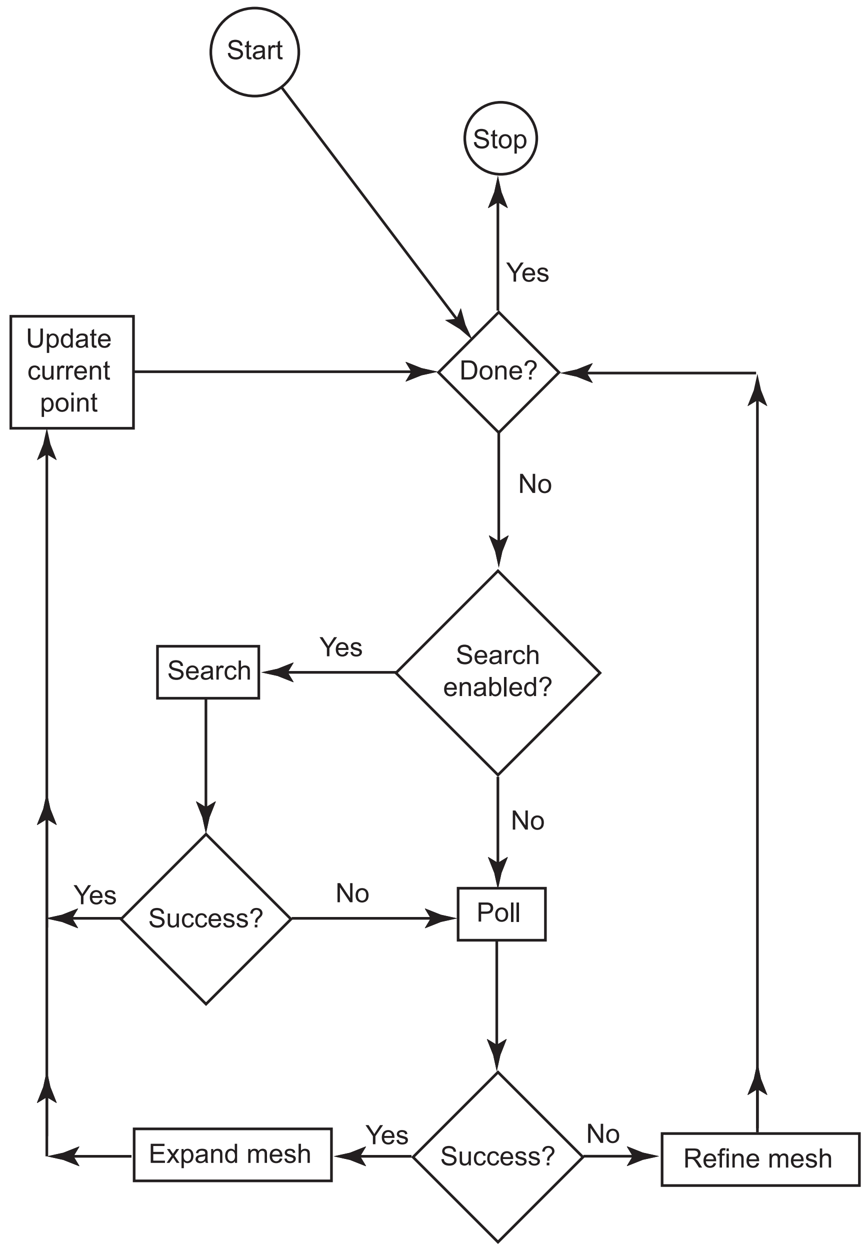
In this section...
"Definition of Search" on page 4-26
"How to Use a Search Method" on page 4-28
"Search Types" on page 4-29
"When to Use Search" on page 4-29

Definition of Search

In `patternsearch`, a *search* is an algorithm that runs before a poll. The search attempts to locate a better point than the current point. (Better means one with lower objective function value.) If the search finds a better point, the better point becomes the current point, and no polling is done at that iteration. If the search does not find a better point, `patternsearch` performs a poll.

By default, `patternsearch` does not use search. To search, see "How to Use a Search Method" on page 4-28.

The figure patternsearch With a Search Method contains a flow chart of direct search including using a search method.



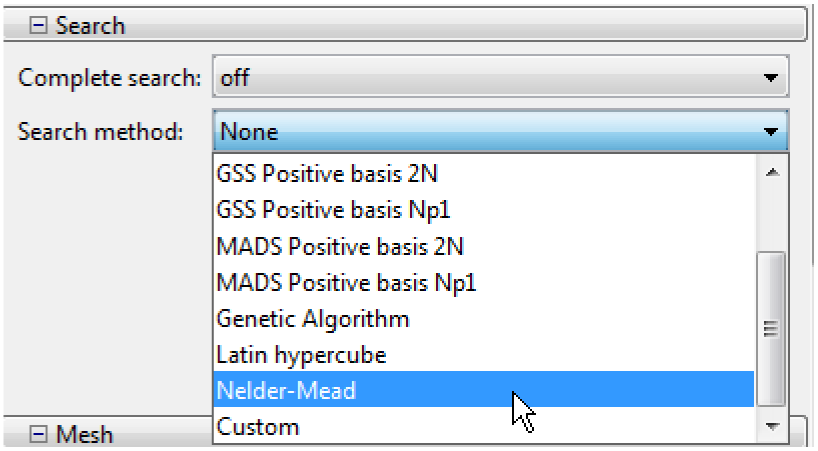
patternsearch With a Search Method

Iteration limit applies to all built-in search methods except those that are poll methods. If you select an iteration limit for the search method, the search is enabled until the iteration limit is reached. Afterwards, `patternsearch` stops searching and only polls.

How to Use a Search Method

To use search in `patternsearch`:

- In Optimization app, choose a **Search method** in the **Search** pane.



Complete search: off

Search method: None

- At the command line, create an options structure with a search method using `psoptimset`. For example, to use Latin hypercube search:

  ```
  opts = psoptimset('SearchMethod',@searchlhs);
  ```

  For more information, including a list of all built-in search methods, consult the `psoptimset` function reference page, and the “Search Options” on page 10-13 section of the options reference.

You can write your own search method. Use the syntax described in “Structure of the Search Function” on page 10-16. To use your search method in a pattern search, give its function handle as the `Custom` **Function** (`SearchMethod`) option.

Search Types

- Poll methods — You can use any poll method as a search algorithm. `patternsearch` conducts one poll step as a search. For this type of search to be beneficial, your search type should be different from your poll type. (`patternsearch` does not search if the selected search method is the same as the poll type.) Therefore, use a MADS search with a GSS or GPS poll, or use a GSS or GPS search with a MADS poll.
- `fminsearch`, also called Nelder-Mead — `fminsearch` is for unconstrained problems only. `fminsearch` runs to its natural stopping criteria; it does not take just one step. Therefore, use `fminsearch` for just one iteration. This is the default setting. To change settings, see “Search Options” on page 10-13.
- `ga` — `ga` runs to its natural stopping criteria; it does not take just one step. Therefore, use `ga` for just one iteration. This is the default setting. To change settings, see “Search Options” on page 10-13.
- Latin hypercube search — Described in “Search Options” on page 10-13. By default, searches 15*n* points, where *n* is the number of variables, and only searches during the first iteration. To change settings, see “Search Options” on page 10-13.

When to Use Search

There are two main reasons to use a search method:

- To speed an optimization (see “Search Methods for Increased Speed” on page 4-29)
- To obtain a better local solution, or to obtain a global solution (see “Search Methods for Better Solutions” on page 4-30)

Search Methods for Increased Speed

Generally, you do not know beforehand whether a search method speeds an optimization or not. So try a search method when:

- You are performing repeated optimizations on similar problems, or on the same problem with different parameters.
- You can experiment with different search methods to find a lower solution time.

Search does not always speed an optimization. For one example where it does, see “Search and Poll” on page 4-32.

Search Methods for Better Solutions

Since search methods run before poll methods, using search can be equivalent to choosing a different starting point for your optimization. This comment holds for the Nelder-Mead, `ga`, and Latin hypercube search methods, all of which, by default, run once at the beginning of an optimization. `ga` and Latin hypercube searches are stochastic, and can search through several basins of attraction.

Setting Solver Tolerances

Tolerance refers to how small a parameter, such a mesh size, can become before the search is halted or changed in some way. You can specify the value of the following tolerances:

- **Mesh tolerance** — When the current mesh size is less than the value of **Mesh tolerance**, the algorithm halts.
- **X tolerance** — After a successful poll, if the distance from the previous best point to the current best point is less than the value of **X tolerance**, the algorithm halts.
- **Function tolerance** — After a successful poll, if the difference between the function value at the previous best point and function value at the current best point is less than the value of **Function tolerance**, the algorithm halts.
- **Nonlinear constraint tolerance** — The algorithm treats a point to be feasible if constraint violation is less than `TolCon`.
- **Bind tolerance** — Bind tolerance applies to linearly constrained problems. It specifies how close a point must get to the boundary of the feasible region before a linear constraint is considered to be active. When a linear constraint is active, the pattern search polls points in directions parallel to the linear constraint boundary as well as the mesh points.

  Usually, you should set **Bind tolerance** to be at least as large as the maximum of **Mesh tolerance**, **X tolerance**, and **Function tolerance**.

Search and Poll

In this section...

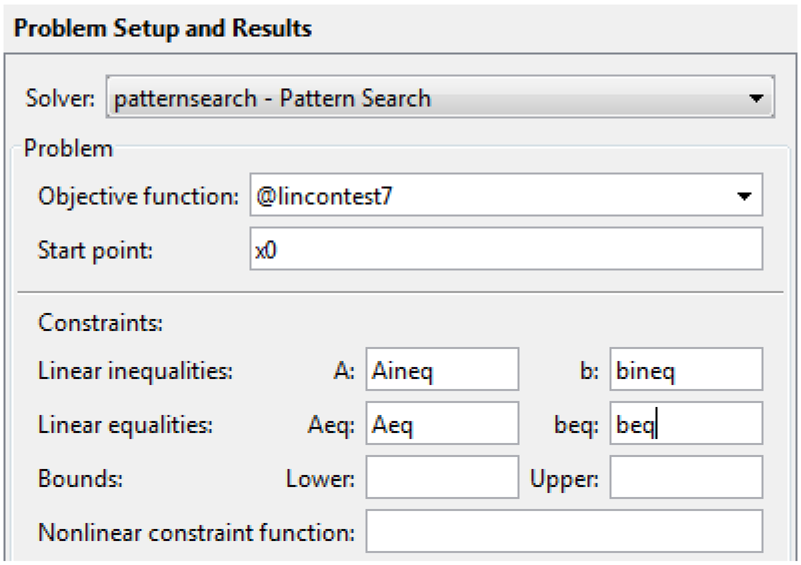
Using a Search Method

In addition to polling the mesh points, the pattern search algorithm can perform an optional step at every iteration, called *search*. At each iteration, the search step applies another optimization method to the current point. If this search does not improve the current point, the poll step is performed.

The following example illustrates the use of a search method on the problem described in "Linearly Constrained Problem" on page 4-69. To set up the example, enter the following commands at the MATLAB prompt to define the initial point and constraints.

```
x0 = [2 1 0 9 1 0];
Aineq = [-8 7 3 -4 9 0];
bineq = 7;
Aeq = [7 1 8 3 3 3; 5 0 -5 1 -5 8; -2 -6 7 1 1 9; 1 -1 2 -2 3 -3];
beq = [84 62 65 1];
```

Then enter the settings shown in the following figures in the Optimization app.



Problem Setup and Results

Solver: patternsearch - Pattern Search

Problem

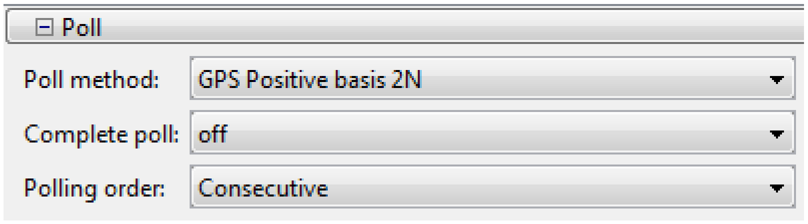
Objective function: @lincontest7

Start point: x0

Constraints:

Linear inequalities:	A:	Aineq	b:	bineq
Linear equalities:	Aeq:	Aeq	beq:	beq
Bounds:	Lower:		Upper:	

Nonlinear constraint function:

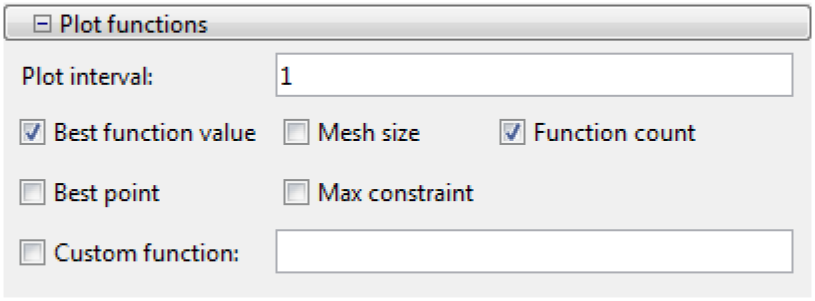


⊟ Poll

Poll method: GPS Positive basis 2N

Complete poll: off

Polling order: Consecutive



⊟ Plot functions

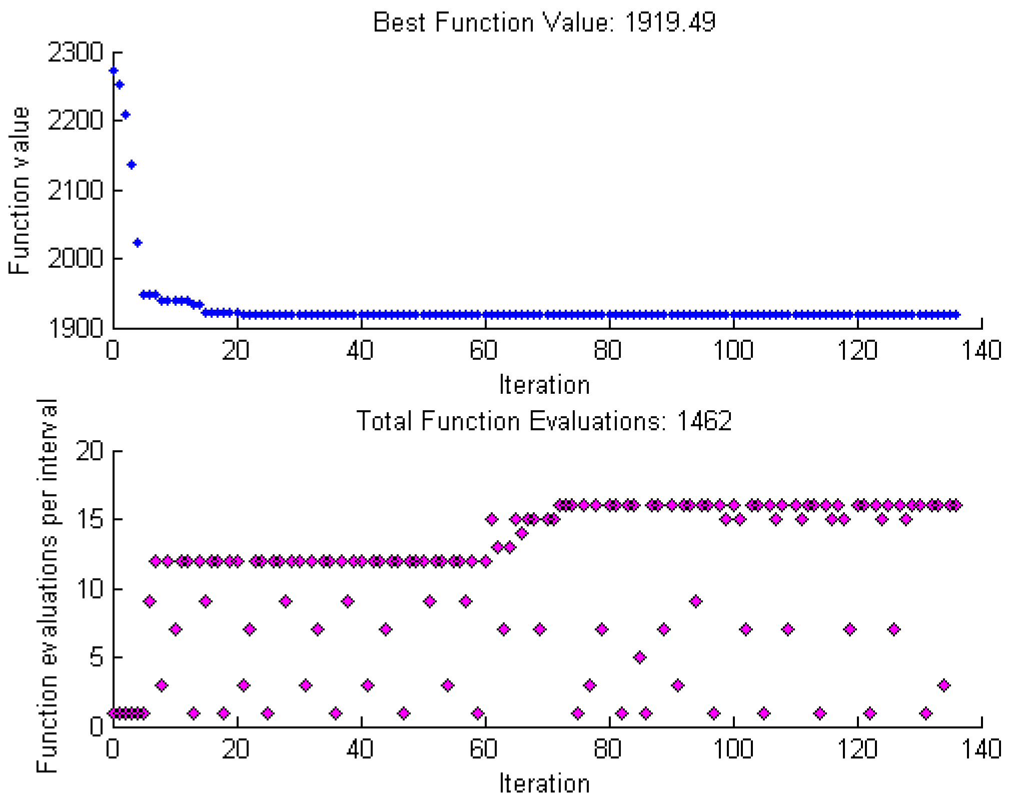
Plot interval: 1

☑ Best function value ☐ Mesh size ☑ Function count

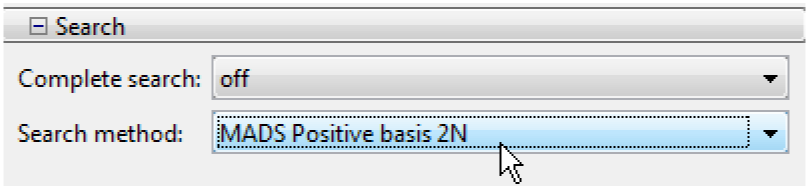
☐ Best point ☐ Max constraint

☐ Custom function:

For comparison, click **Start** to run the example without a search method. This displays the plots shown in the following figure.



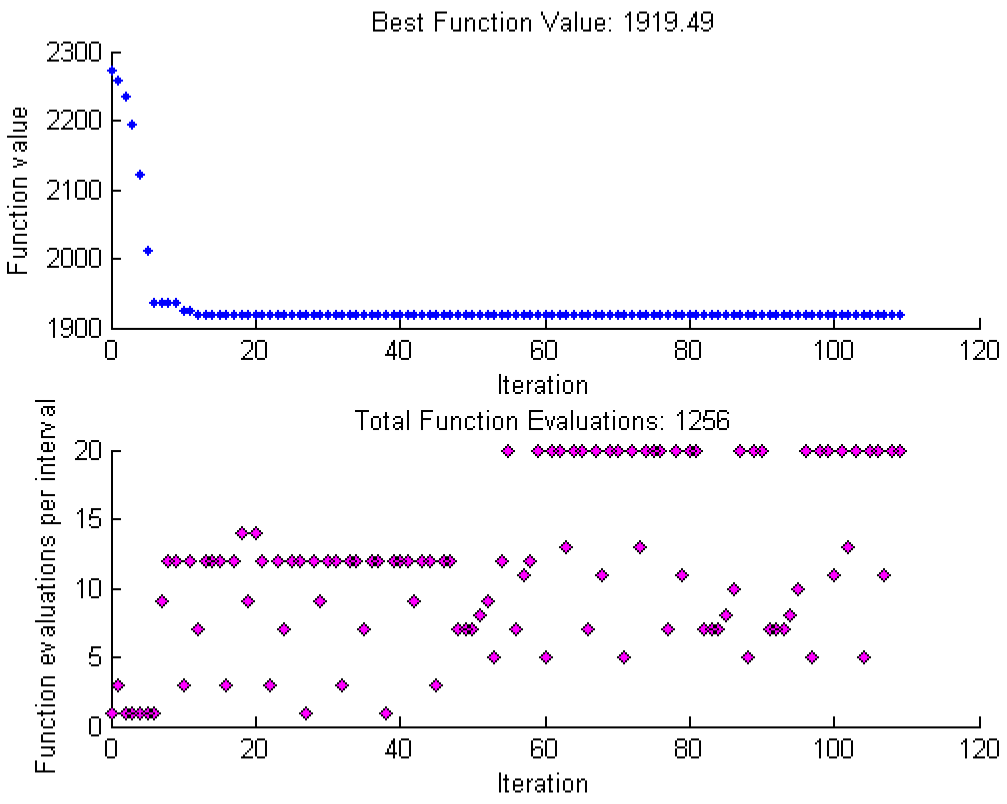
To see the effect of using a search method, select `MADS Positive Basis 2N` in the **Search method** field in **Search** options.



Complete search: off

Search method: MADS Positive basis 2N

This sets the search method to be a pattern search using the pattern for `MADS Positive Basis 2N`. Then click **Start** to run the pattern search. This displays the following plots.



Note that using the search method reduces the total function evaluations—from 1206 to 1159—and reduces the number of iterations from 106 to 97.

Search Using a Different Solver

`patternsearch` takes a long time to minimize Rosenbrock's function. The function is

$$f(x) = 100\left(x_2 - x_1^2\right)^2 + (1 - x_1)^2 .$$

Rosenbrock's function is described and plotted in "Solve a Constrained Nonlinear Problem" in the Optimization Toolbox documentation. The `dejong2fcn.m` file, which in included in the toolbox, calculates this function.

1 Set `patternsearch` options to `MaxFunEvals` = `5000` and `MaxIter` = `2000`:

```
opts = psoptimset('MaxFunEvals',5000,'MaxIter',2000);
```

2 Run `patternsearch` starting from `[-1.9 2]`:

```
[x feval eflag output] = patternsearch(@dejong2fcn,...
    [-1.9,2],[],[],[],[],[],[],[],opts);

Maximum number of function evaluations exceeded:
increase options.MaxFunEvals.

feval

feval =
    0.8560
```

The optimization did not complete, and the result is not very close to the optimal value of 0.

3 Set the options to use `fminsearch` as the search method:

```
opts = psoptimset(opts,'SearchMethod',@searchneldermead);
```

4 Rerun the optimization, the results are much better:

```
[x2 feval2 eflag2 output2] = patternsearch(@dejong2fcn,...
    [-1.9,2],[],[],[],[],[],[],[],opts);
Optimization terminated: mesh size less than options.TolMesh.

feval2

feval2 =
  4.0686e-010
```

`fminsearch` is not as closely tied to coordinate directions as the default GPS `patternsearch` poll method. Therefore, `fminsearch` is more efficient at getting close to the minimum of Rosenbrock's function. Adding the search method in this case is effective.

Nonlinear Constraint Solver Algorithm

The pattern search algorithm uses the Augmented Lagrangian Pattern Search (ALPS) algorithm to solve nonlinear constraint problems. The optimization problem solved by the ALPS algorithm is

$$\min_{x} f(x)$$

such that

$$\begin{aligned} c_i(x) &\le 0,\ i = 1 \ldots m \\ ceq_i(x) &= 0,\ i = m+1 \ldots mt \\ A \cdot x &\le b \\ Aeq \cdot x &= beq \\ lb \le x &\le ub, \end{aligned}$$

where $c(x)$ represents the nonlinear inequality constraints, $ceq(x)$ represents the equality constraints, m is the number of nonlinear inequality constraints, and mt is the total number of nonlinear constraints.

The ALPS algorithm attempts to solve a nonlinear optimization problem with nonlinear constraints, linear constraints, and bounds. In this approach, bounds and linear constraints are handled separately from nonlinear constraints. A subproblem is formulated by combining the objective function and nonlinear constraint function using the Lagrangian and the penalty parameters. A sequence of such optimization problems are approximately minimized using a pattern search algorithm such that the linear constraints and bounds are satisfied.

Each subproblem solution represents one iteration. The number of function evaluations per iteration is therefore much higher when using nonlinear constraints than otherwise.

A subproblem formulation is defined as

$$\Theta(x,\lambda,s,\rho) = f(x) - \sum_{i=1}^{m} \lambda_i s_i \log(s_i - c_i(x)) + \sum_{i=m+1}^{mt} \lambda_i ceq_i(x) + \frac{\rho}{2} \sum_{i=m+1}^{mt} ceq_i(x)^2,$$

where

- The components λ_i of the vector λ are nonnegative and are known as Lagrange multiplier estimates
- The elements s_i of the vector s are nonnegative shifts
- ρ is the positive penalty parameter.

The algorithm begins by using an initial value for the penalty parameter (`InitialPenalty`).

The pattern search minimizes a sequence of subproblems, each of which is an approximation of the original problem. Each subproblem has a fixed value of λ, s, and ρ. When the subproblem is minimized to a required accuracy and satisfies feasibility conditions, the Lagrangian estimates are updated. Otherwise, the penalty parameter is increased by a penalty factor (`PenaltyFactor`). This results in a new subproblem formulation and minimization problem. These steps are repeated until the stopping criteria are met.

Each subproblem solution represents one iteration. The number of function evaluations per iteration is therefore much higher when using nonlinear constraints than otherwise.

For a complete description of the algorithm, see the following references:

References

[1] Kolda, Tamara G., Robert Michael Lewis, and Virginia Torczon. “A generating set direct search augmented Lagrangian algorithm for optimization with a combination of general and linear constraints.” Technical Report SAND2006-5315, Sandia National Laboratories, August 2006.

[2] Conn, A. R., N. I. M. Gould, and Ph. L. Toint. “A Globally Convergent Augmented Lagrangian Algorithm for Optimization with General Constraints and Simple Bounds,” *SIAM Journal on Numerical Analysis*, Volume 28, Number 2, pages 545–572, 1991.

[3] Conn, A. R., N. I. M. Gould, and Ph. L. Toint. “A Globally Convergent Augmented Lagrangian Barrier Algorithm for Optimization with General Inequality Constraints and Simple Bounds,” *Mathematics of Computation*, Volume 66, Number 217, pages 261–288, 1997.

Custom Plot Function

In this section...

About Custom Plot Functions

To use a plot function other than those included with the software, you can write your own custom plot function that is called at each iteration of the pattern search to create the plot. This example shows how to create a plot function that displays the logarithmic change in the best objective function value from the previous iteration to the current iteration. More plot function details are available in "Plot Options" on page 10-29.

Creating the Custom Plot Function

To create the plot function for this example, copy and paste the following code into a new function file in the MATLAB Editor:

```
function stop = psplotchange(optimvalues, flag)
% PSPLOTCHANGE Plots the change in the best objective function
% value from the previous iteration.

% Best objective function value in the previous iteration
persistent last_best

stop = false;
if(strcmp(flag,'init'))
        set(gca,'Yscale','log'); %Set up the plot
        hold on;
        xlabel('Iteration');
        ylabel('Log Change in Values');
        title(['Change in Best Function Value']);
end

% Best objective function value in the current iteration
best = min(optimvalues.fval);

 % Set last_best to best
if optimvalues.iteration == 0
last_best = best;
```

```
else
        %Change in objective function value
    change = last_best - best;
        plot(optimvalues.iteration, change, '.r');
end
```

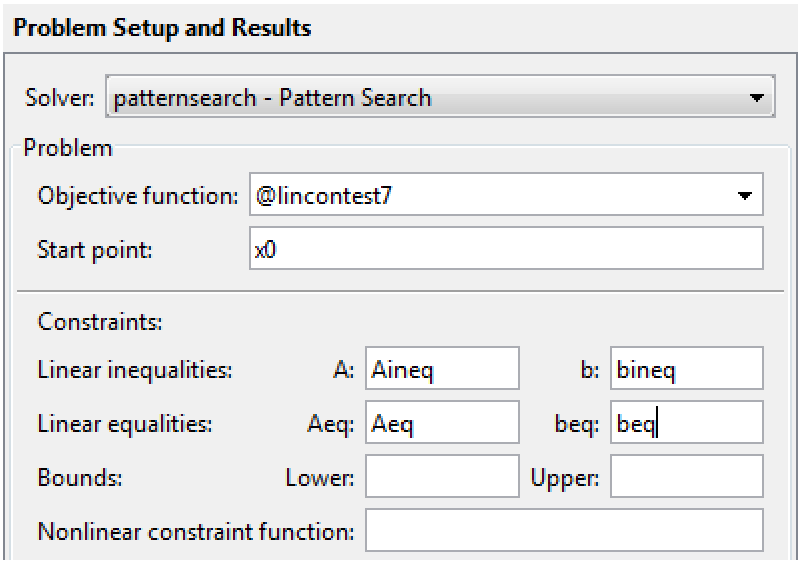
Then save the file as `psplotchange.m` in a folder on the MATLAB path.

Setting Up the Problem

The problem is the same as "Linearly Constrained Problem" on page 4-69. To set up the problem:

1. Enter the following at the MATLAB command line:

   ```
   x0 = [2 1 0 9 1 0];
   Aineq = [-8 7 3 -4 9 0];
   bineq = 7;
   Aeq = [7 1 8 3 3 3; 5 0 -5 1 -5 8; -2 -6 7 1 1 9; 1 -1 2 -2 3 -3];
   beq = [84 62 65 1];
   ```
2. Enter `optimtool` to open the Optimization app.
3. Choose the `patternsearch` solver.
4. Set up the problem to match the following figure.



Problem Setup and Results

Solver: patternsearch - Pattern Search

Problem

Objective function: @lincontest7

Start point: x0

Constraints:

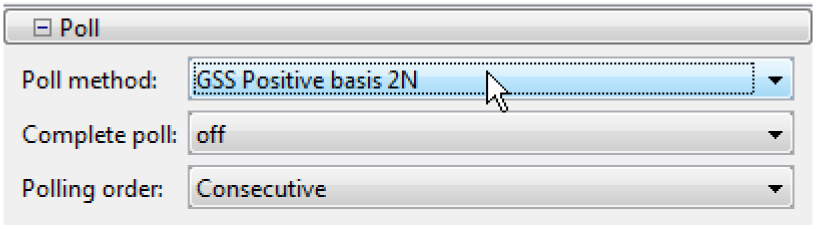
Linear inequalities: A: Aineq b: bineq

Linear equalities: Aeq: Aeq beq: beq

Bounds: Lower: Upper:

Nonlinear constraint function:

5 Since this is a linearly constrained problem, set the **Poll method** to `GSS Positive basis 2N`.



Poll

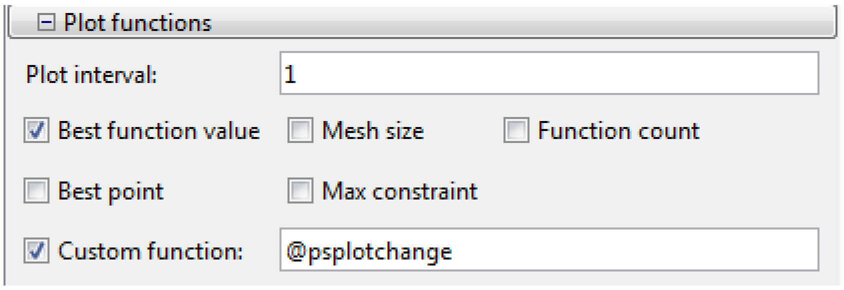
Poll method: GSS Positive basis 2N

Complete poll: off

Polling order: Consecutive

Using the Custom Plot Function

To use the custom plot function, select **Custom function** in the **Plot functions** pane and enter `@psplotchange` in the field to the right. To compare the custom plot with the best function value plot, also select **Best function value**.



Plot functions

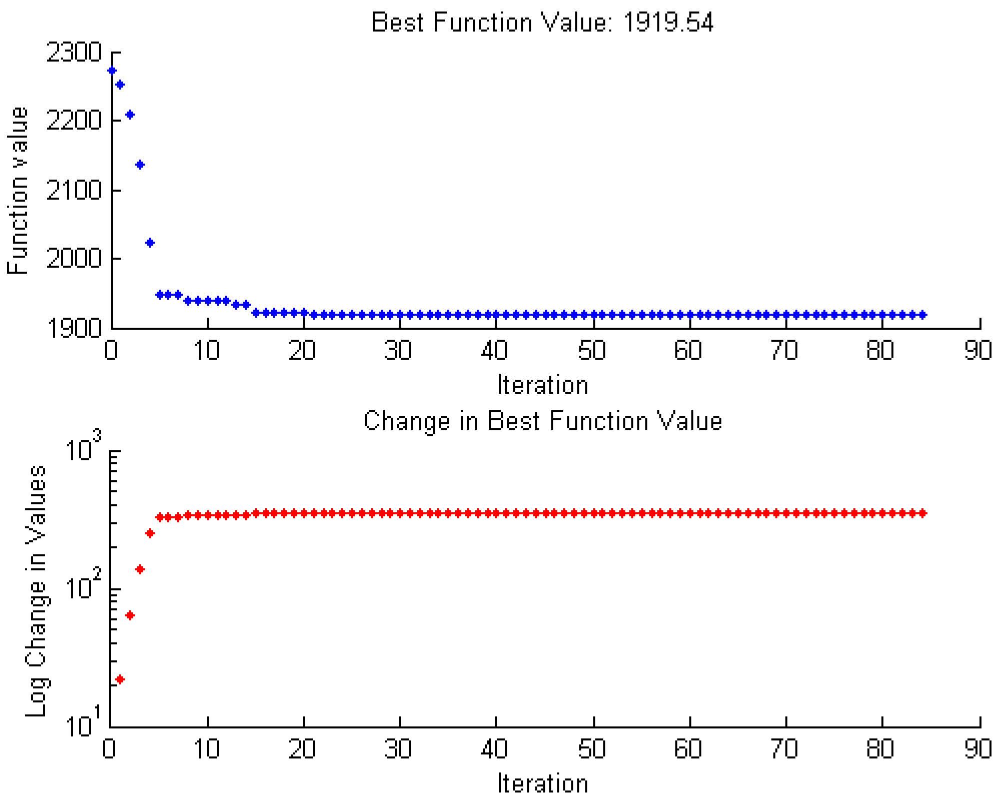
Plot interval: 1

[✓] Best function value [ ] Mesh size [ ] Function count

[ ] Best point [ ] Max constraint

[✓] Custom function: @psplotchange

Now, when you run the example, the pattern search tool displays the plots shown in the following figure.



Note that because the scale of the *y*-axis in the lower custom plot is logarithmic, the plot will only show changes that are greater than 0.

How the Plot Function Works

The plot function uses information contained in the following structures, which the Optimization app passes to the function as input arguments:

- `optimvalues` — Structure containing the current state of the solver
- `flag` — String indicating the current status of the algorithm

The most important statements of the custom plot function, `psplotchange.m`, are summarized in the following table.

Custom Plot Function Statements

Statement	Description
`persistent last_best`	Creates the persistent variable `last_best`, the best objective function value in the previous generation. Persistent variables are preserved over multiple calls to the plot function.
`set(gca,'Yscale','log')`	Sets up the plot before the algorithm starts.
`best = min(optimvalues.fval)`	Sets `best` equal to the minimum objective function value. The field `optimvalues.fval` contains the objective function value in the current iteration. The variable `best` is the minimum objective function value. For a complete description of the fields of the structure `optimvalues`, see "Structure of the Plot Functions" on page 10-11.
`change = last_best - best`	Sets the variable `change` to the best objective function value at the previous iteration minus the best objective function value in the current iteration.
`plot(optimvalues.iteration, change, '.r')`	Plots the variable `change` at the current objective function value, for the current iteration contained in`optimvalues.iteration`.

Set Options

In this section...
"Set Options Using psoptimset" on page 4-44
"Create Options and Problems Using the Optimization App" on page 4-46

Set Options Using psoptimset

You can specify any available `patternsearch` options by passing an `options` structure as an input argument to `patternsearch` using the syntax

```
[x fval] = patternsearch(@fitnessfun,nvars, ...
            A,b,Aeq,beq,lb,ub,nonlcon,options)
```

Pass in empty brackets `[]` for any constraints that do not appear in the problem.

You create the `options` structure using the function `psoptimset`.

```
options = psoptimset(@patternsearch)
```

This returns the `options` structure with the default values for its fields.

```
options =
            TolMesh: 1.0000e-06
             TolCon: 1.0000e-06
               TolX: 1.0000e-06
             TolFun: 1.0000e-06
            TolBind: 1.0000e-03
            MaxIter: '100*numberofvariables'
        MaxFunEvals: '2000*numberofvariables'
          TimeLimit: Inf
    MeshContraction: 0.5000
      MeshExpansion: 2
    MeshAccelerator: 'off'
         MeshRotate: 'on'
    InitialMeshSize: 1
          ScaleMesh: 'on'
        MaxMeshSize: Inf
     InitialPenalty: 10
      PenaltyFactor: 100
         PollMethod: 'gpspositivebasis2n'
       CompletePoll: 'off'
```

```
        PollingOrder: 'consecutive'
        SearchMethod: []
      CompleteSearch: 'off'
             Display: 'final'
          OutputFcns: []
            PlotFcns: []
        PlotInterval: 1
               Cache: 'off'
           CacheSize: 10000
            CacheTol: 2.2204e-16
          Vectorized: 'off'
         UseParallel: 0
```

The `patternsearch` function uses these default values if you do not pass in `options` as an input argument.

The value of each option is stored in a field of the `options` structure, such as `options.MeshExpansion`. You can display any of these values by entering `options` followed by the name of the field. For example, to display the mesh expansion factor for the pattern search, enter

```
options.MeshExpansion

ans =
    2
```

To create an `options` structure with a field value that is different from the default, use `psoptimset`. For example, to change the mesh expansion factor to `3` instead of its default value `2`, enter

```
options = psoptimset('MeshExpansion',3);
```

This creates the `options` structure with all values set to empty except for `MeshExpansion`, which is set to `3`. (An empty field causes `patternsearch` to use the default value.)

If you now call `patternsearch` with the argument `options`, the pattern search uses a mesh expansion factor of `3`.

If you subsequently decide to change another field in the `options` structure, such as setting `PlotFcns` to `@psplotmeshsize`, which plots the mesh size at each iteration, call `psoptimset` with the syntax

```
options = psoptimset(options, 'PlotFcns', @psplotmeshsize)
```

This preserves the current values of all fields of `options` except for `PlotFcns`, which is changed to `@plotmeshsize`. Note that if you omit the `options` input argument, `psoptimset` resets `MeshExpansion` to its default value, which is `2`.

You can also set both `MeshExpansion` and `PlotFcns` with the single command

```
options = psoptimset('MeshExpansion',3,'PlotFcns',@plotmeshsize)
```

Create Options and Problems Using the Optimization App

As an alternative to creating the options structure using `psoptimset`, you can set the values of options in the Optimization app and then export the options to a structure in the MATLAB workspace, as described in the "Importing and Exporting Your Work" section of the Optimization Toolbox documentation. If you export the default options in the Optimization app, the resulting `options` structure has the same settings as the default structure returned by the command

```
options = psoptimset
```

except for the default value of `'Display'`, which is `'final'` when created by `psoptimset`, but `'none'` when created in the Optimization app.

You can also export an entire problem from the Optimization app and run it from the command line. Enter

```
patternsearch(problem)
```

where `problem` is the name of the exported problem.

Polling Types

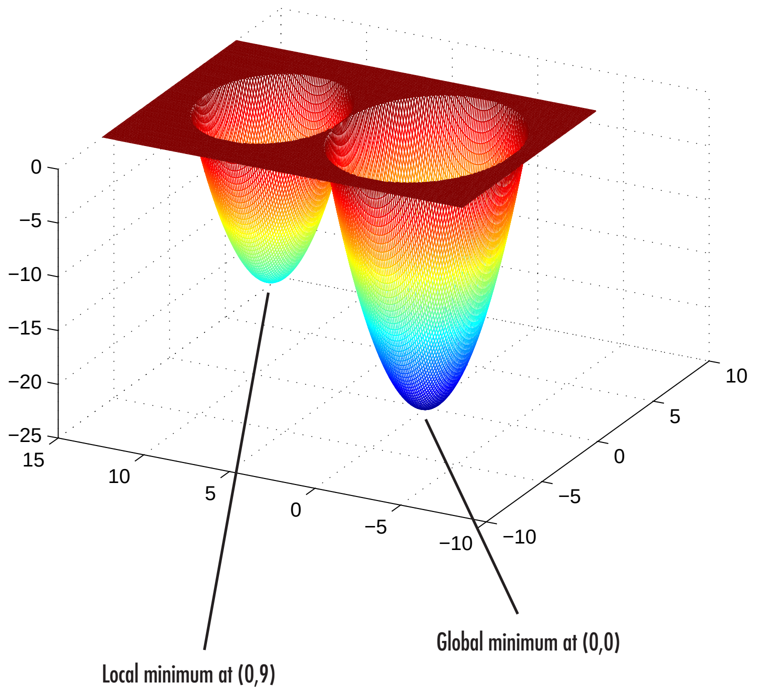
In this section...

Using a Complete Poll in a Generalized Pattern Search

As an example, consider the following function.

$$f(x_1, x_2) = \begin{cases} x_1^2 + x_2^2 - 25 & \text{for } x_1^2 + x_2^2 \le 25 \\ x_1^2 + (x_2 - 9)^2 - 16 & \text{for } x_1^2 + (x_2 - 9)^2 \le 16 \\ 0 & \text{otherwise.} \end{cases}$$

The following figure shows a plot of the function.



The global minimum of the function occurs at (0, 0), where its value is -25. However, the function also has a local minimum at (0, 9), where its value is -16.

To create a file that computes the function, copy and paste the following code into a new file in the MATLAB Editor.

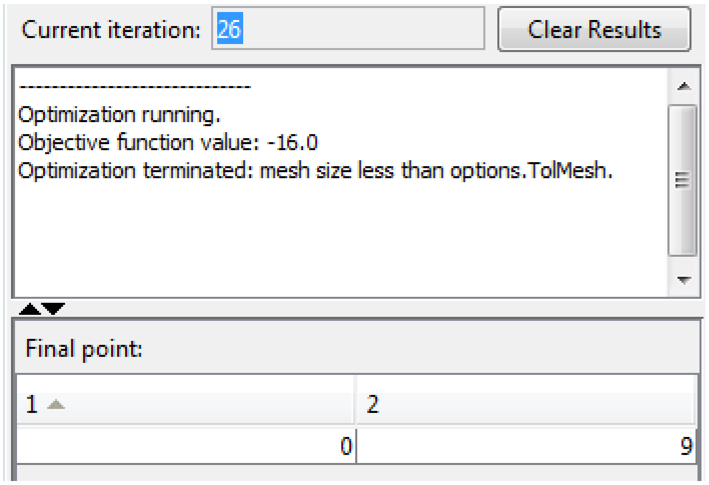
```
function z = poll_example(x)
if x(1)^2 + x(2)^2 <= 25
    z = x(1)^2 + x(2)^2 - 25;
elseif x(1)^2 + (x(2) - 9)^2 <= 16
    z = x(1)^2 + (x(2) - 9)^2 - 16;
else z = 0;
end
```

Then save the file as `poll_example.m` in a folder on the MATLAB path.

To run a pattern search on the function, enter the following in the Optimization app:

- Set **Solver** to `patternsearch`.
- Set **Objective function** to `@poll_example`.
- Set **Start point** to `[0 5]`.
- Set **Level of display** to `Iterative` in the **Display to command window** options.

Click **Start** to run the pattern search with **Complete poll** set to `Off`, its default value. The Optimization app displays the results in the **Run solver and view results** pane, as shown in the following figure.



```
-----------------------------
Optimization running.
Objective function value: -16.0
Optimization terminated: mesh size less than options.TolMesh.
```

1	2
0	9

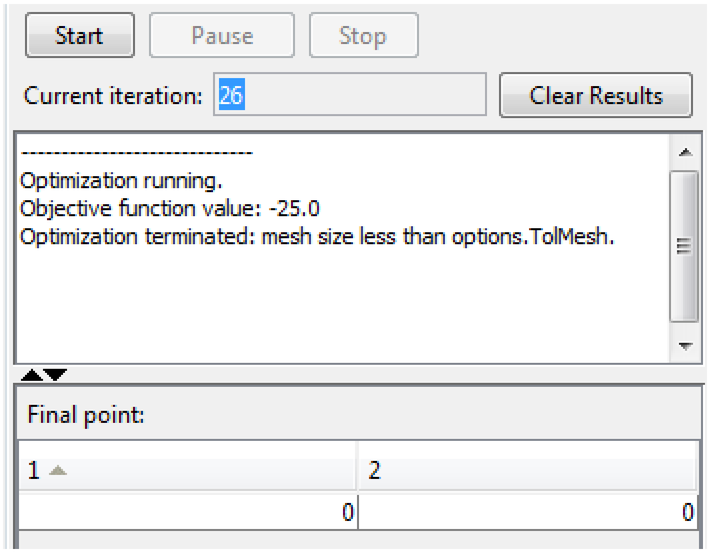
The pattern search returns the local minimum at (0, 9). At the initial point, (0, 5), the objective function value is 0. At the first iteration, the search polls the following mesh points.
$f((0, 5) + (1, 0)) = f(1, 5) = 0$
$f((0, 5) + (0, 1)) = f(0, 6) = -7$

As soon as the search polls the mesh point (0, 6), at which the objective function value is less than at the initial point, it stops polling the current mesh and sets the current point at the next iteration to (0, 6). Consequently, the search moves toward the local minimum at (0, 9) at the first iteration. You see this by looking at the first two lines of the command line display.

```
Iter     f-count     f(x)      MeshSize     Method
    0         1        0             1
    1         3       -7             2     Successful Poll
```

Note that the pattern search performs only two evaluations of the objective function at the first iteration, increasing the total function count from 1 to 3.

Next, set **Complete poll** to `On` and click **Start**. The **Run solver and view results** pane displays the following results.



Current iteration: 26

```
----------------------------
Optimization running.
Objective function value: -25.0
Optimization terminated: mesh size less than options.TolMesh.
```

Final point:

1	2
0	0

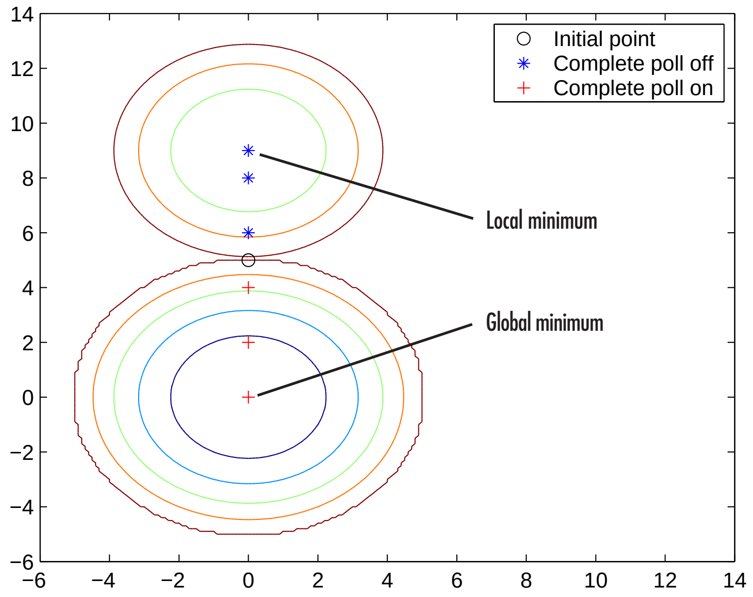
This time, the pattern search finds the global minimum at (0, 0). The difference between this run and the previous one is that with **Complete poll** set to `On`, at the first iteration the pattern search polls all four mesh points.
$f((0, 5) + (1, 0)) = f(1, 5) = 0$
$f((0, 5) + (0, 1)) = f(0, 6) = -6$
$f((0, 5) + (-1, 0)) = f(-1, 5) = 0$
$f((0, 5) + (0, -1)) = f(0, 4) = -9$

Because the last mesh point has the lowest objective function value, the pattern search selects it as the current point at the next iteration. The first two lines of the command-line display show this.

```
Iter     f-count      f(x)      MeshSize     Method
    0          1         0             1
    1          5        -9             2    Successful Poll
```

In this case, the objective function is evaluated four times at the first iteration. As a result, the pattern search moves toward the global minimum at (0, 0).

The following figure compares the sequence of points returned when **Complete poll** is set to `Off` with the sequence when **Complete poll** is `On`.



Compare the Efficiency of Poll Options

This example shows how several poll options interact in terms of iterations and total function evaluations. The main results are:

- GSS is more efficient than GPS or MADS for linearly constrained problems.
- Whether setting `CompletePoll` to `'on'` increases efficiency or decreases efficiency is unclear, although it affects the number of iterations.
- Similarly, whether having a `2N` poll is more or less efficient than having an `Np1` poll is also unclear. The most efficient poll is `GSS Positive Basis Np1` with **Complete poll** set to `on`. The least efficient is `MADS Positive Basis Np1` with **Complete poll** set to `on`.

Note: The efficiency of an algorithm depends on the problem. GSS is efficient for linearly constrained problems. However, predicting the efficiency implications of the other poll options is difficult, as is knowing which poll type works best with other constraints.

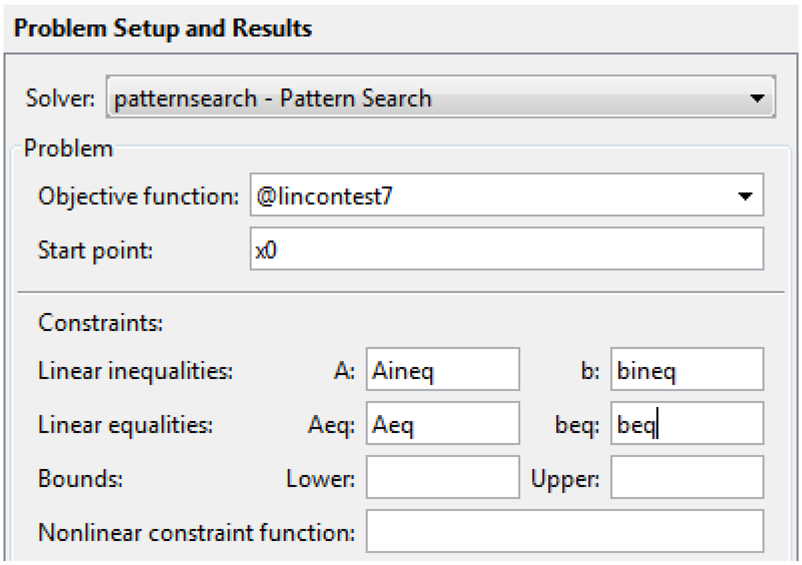
Problem setup

The problem is the same as in "Performing a Pattern Search on the Example" on page 4-70. This linearly constrained problem uses the `lincontest7` file that comes with the toolbox:

1. Enter the following into your MATLAB workspace:

```
x0 = [2 1 0 9 1 0];
Aineq = [-8 7 3 -4 9 0];
bineq = 7;
Aeq = [7 1 8 3 3 3; 5 0 -5 1 -5 8; -2 -6 7 1 1 9; 1 -1 2 -2 3 -3];
beq = [84 62 65 1];
```

2. Open the Optimization app by entering `optimtool` at the command line.
3. Choose the `patternsearch` solver.
4. Enter the problem and constraints as pictured.


Problem Setup and Results

Solver: patternsearch - Pattern Search

Problem

Objective function: @lincontest7

Start point: x0

Constraints:

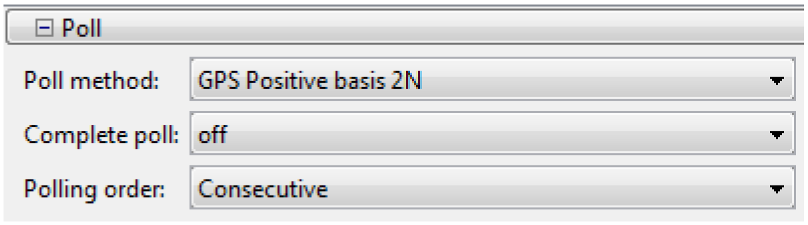
Linear inequalities: A: Aineq b: bineq

Linear equalities: Aeq: Aeq beq: beq

Bounds: Lower: Upper:

Nonlinear constraint function:

5. Ensure that the **Poll method** is `GPS Positive basis 2N`.


Poll

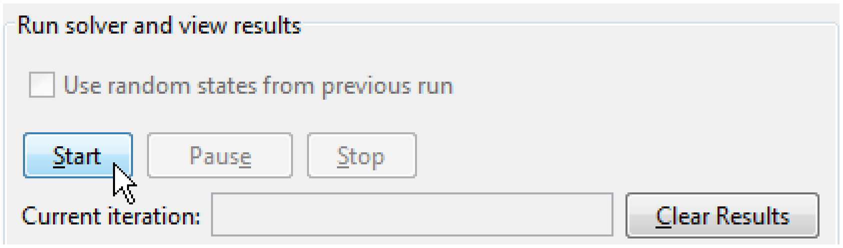
Poll method: GPS Positive basis 2N

Complete poll: off

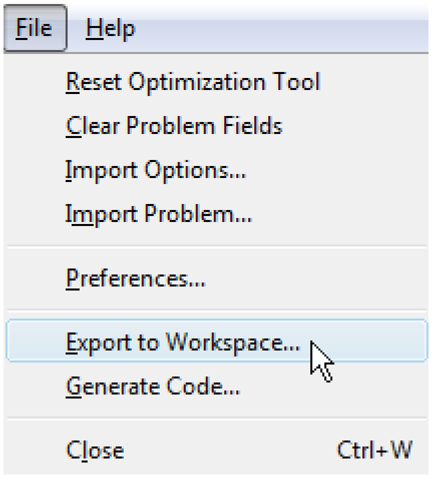
Polling order: Consecutive

Generate the Results

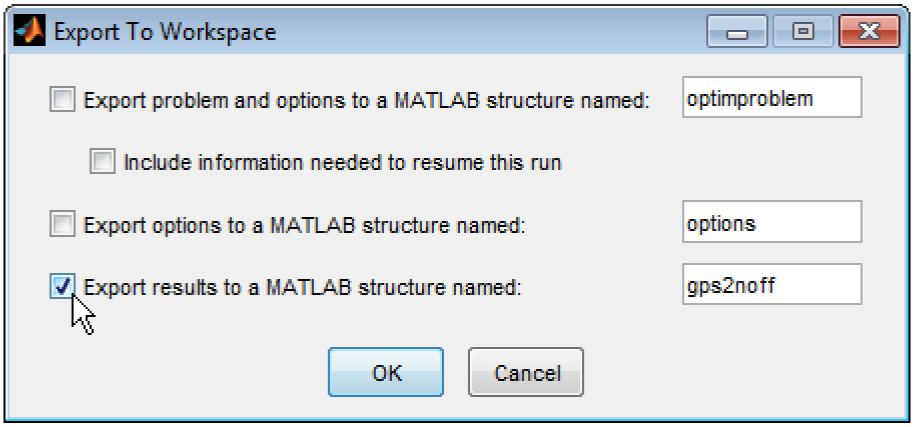
1 Run the optimization.


Run solver and view results
Use random states from previous run
Start
Pause
Stop
Current iteration:
Clear Results

2 Choose **File > Export to Workspace**.



3 Export the results to a structure named `gps2noff`.


Export To Workspace

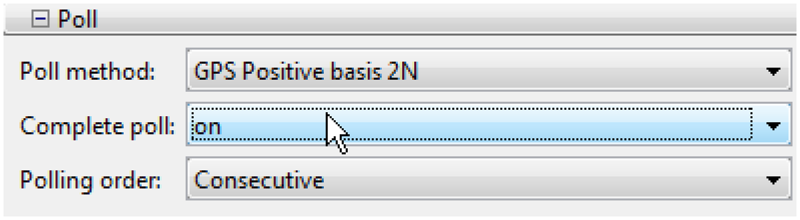
[ ] Export problem and options to a MATLAB structure named: optimproblem

[ ] Include information needed to resume this run

[ ] Export options to a MATLAB structure named: options

[✓] Export results to a MATLAB structure named: gps2noff

4 Set **Options > Poll > Complete poll** to `on`.


Poll

Poll method: GPS Positive basis 2N

Complete poll: on

Polling order: Consecutive

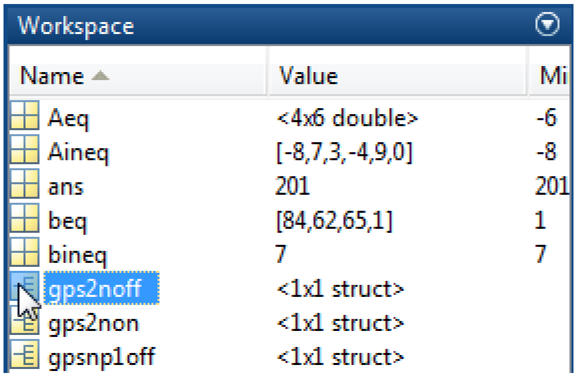
5 Run the optimization.
6 Export the result to a structure named `gps2non`.
7 Set **Options > Poll > Poll method** to `GPS Positive basis Np1` and set **Complete poll** to `off`.
8 Run the optimization.
9 Export the result to a structure named `gpsnp1off`.
10 Set **Complete poll** to `on`.
11 Run the optimization.
12 Export the result to a structure named `gpsnp1on`.
13 Continue in a like manner to create solution structures for the other poll methods with **Complete poll** set `on` and `off`: `gss2noff`, `gss2non`, `gssnp1off`, `gssnp1on`, `mads2noff`, `mads2non`, `madsnp1off`, and `madsnp1on`.

Examine the Results

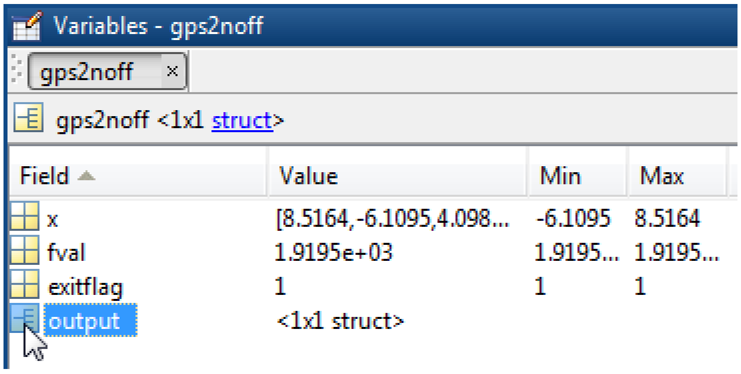
You have the results of 12 optimization runs. The following table shows the efficiency of the runs, measured in total function counts and in iterations. Your MADS results could differ, since MADS is a stochastic algorithm.

Algorithm	Function Count	Iterations
GPS2N, complete poll off	1462	136
GPS2N, complete poll on	1396	96
GPSNp1, complete poll off	864	118
GPSNp1, complete poll on	1007	104
GSS2N, complete poll off	758	84
GSS2N, complete poll on	889	74
GSSNp1, complete poll off	533	94
GSSNp1, complete poll on	491	70
MADS2N, complete poll off	922	162
MADS2N, complete poll on	2285	273
MADSNp1, complete poll off	1155	201
MADSNp1, complete poll on	1651	201

To obtain, say, the first row in the table, enter `gps2noff.output.funccount` and `gps2noff.output.iterations`. You can also examine a structure in the Variables editor by double-clicking the structure in the Workspace Browser, and then double-clicking the `output` structure.


Workspace

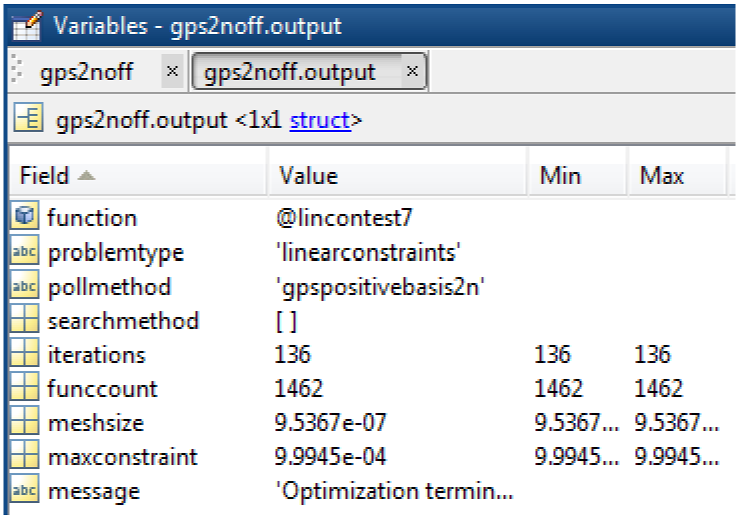
Name	Value	Mi
Aeq	<4x6 double>	-6
Aineq	[-8,7,3,-4,9,0]	-8
ans	201	201
beq	[84,62,65,1]	1
bineq	7	7
gps2noff	<1x1 struct>	
gps2non	<1x1 struct>	
gpsnp1off	<1x1 struct>	



Variables - gps2noff

gps2noff <1x1 struct>

Field	Value	Min	Max
x	[8.5164,-6.1095,4.098...	-6.1095	8.5164
fval	1.9195e+03	1.9195...	1.9195...
exitflag	1	1	1
output	<1x1 struct>		



Variables - gps2noff.output

gps2noff.output <1x1 struct>

Field	Value	Min	Max
function	@lincontest7		
problemtype	'linearconstraints'		
pollmethod	'gpspositivebasis2n'		
searchmethod	[ ]		
iterations	136	136	136
funccount	1462	1462	1462
meshsize	9.5367e-07	9.5367...	9.5367...
maxconstraint	9.9945e-04	9.9945...	9.9945...
message	'Optimization termin...		

The main results gleaned from the table are:

- Setting **Complete poll** to `on` generally lowers the number of iterations for GPS and GSS, but the change in number of function evaluations is unpredictable.
- Setting **Complete poll** to `on` does not necessarily change the number of iterations for MADS, but substantially increases the number of function evaluations.
- The most efficient algorithm/options settings, with efficiency meaning lowest function count:

  1. `GSS Positive basis Np1` with **Complete poll** set to `on` (function count 491)

2. `GSS Positive basis Np1` with **Complete poll** set to `off` (function count 533)
3. `GSS Positive basis 2N` with **Complete poll** set to `off` (function count 758)
4. `GSS Positive basis 2N` with **Complete poll** set to `on` (function count 889)

   The other poll methods had function counts exceeding 900.

- For this problem, the most efficient poll is `GSS Positive Basis Np1` with **Complete poll** set to `on`, although the **Complete poll** setting makes only a small difference. The least efficient poll is `MADS Positive Basis 2N` with **Complete poll** set to `on`. In this case, the **Complete poll** setting makes a substantial difference.

Set Mesh Options

In this section...

Mesh Expansion and Contraction

The **Expansion factor** and **Contraction factor** options, in **Mesh** options, control how much the mesh size is expanded or contracted at each iteration. With the default **Expansion factor** value of `2`, the pattern search multiplies the mesh size by `2` after each successful poll. With the default **Contraction factor** value of `0.5`, the pattern search multiplies the mesh size by `0.5` after each unsuccessful poll.

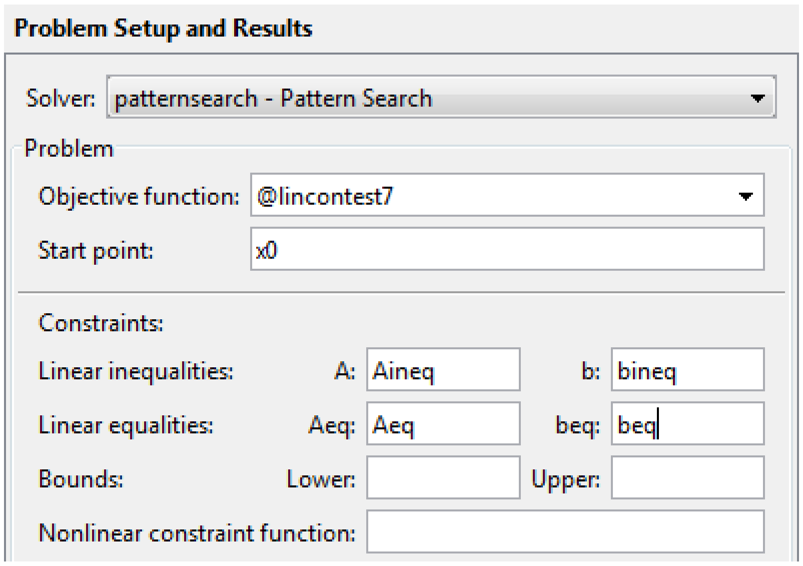
You can view the expansion and contraction of the mesh size during the pattern search by selecting **Mesh size** in the **Plot functions** pane. To also display the values of the mesh size and objective function at the command line, set **Level of display** to `Iterative` in the **Display to command window** options.

For example, set up the problem described in "Linearly Constrained Problem" on page 4-69 as follows:

1 Enter the following at the command line:

```
x0 = [2 1 0 9 1 0];
Aineq = [-8 7 3 -4 9 0];
bineq = 7;
Aeq = [7 1 8 3 3 3; 5 0 -5 1 -5 8; -2 -6 7 1 1 9; 1 -1 2 -2 3 -3];
beq = [84 62 65 1];
```

2 Set up your problem in the Optimization app to match the following figures.



Problem Setup and Results

Solver: patternsearch - Pattern Search

Problem

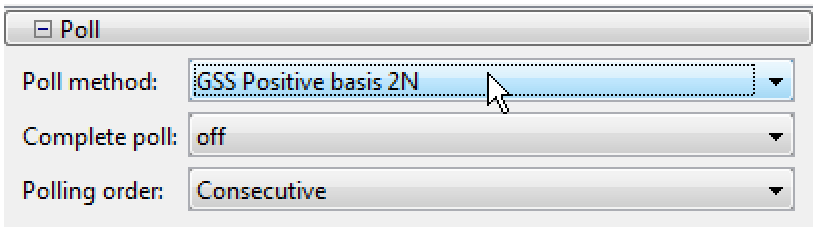
Objective function: @lincontest7

Start point: x0

Constraints:

Linear inequalities:	A:	Aineq	b:	bineq
Linear equalities:	Aeq:	Aeq	beq:	beq
Bounds:	Lower:		Upper:	

Nonlinear constraint function:

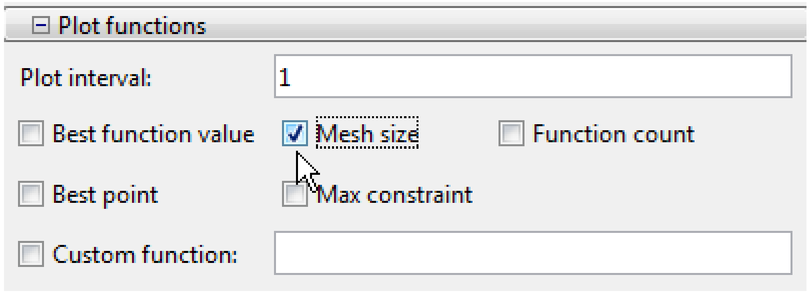


Poll

Poll method: GSS Positive basis 2N

Complete poll: off

Polling order: Consecutive



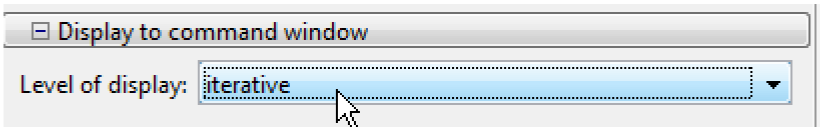
Plot functions

Plot interval: 1

☐ Best function value ✓ Mesh size ☐ Function count

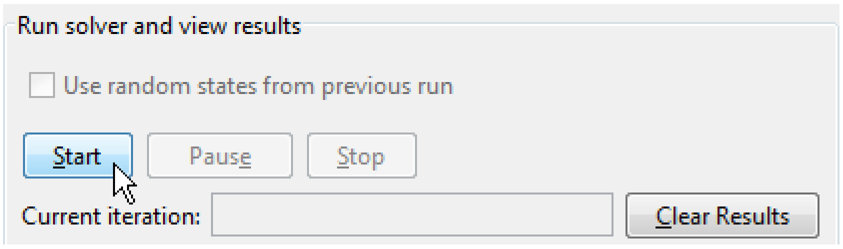
☐ Best point ☐ Max constraint

☐ Custom function:

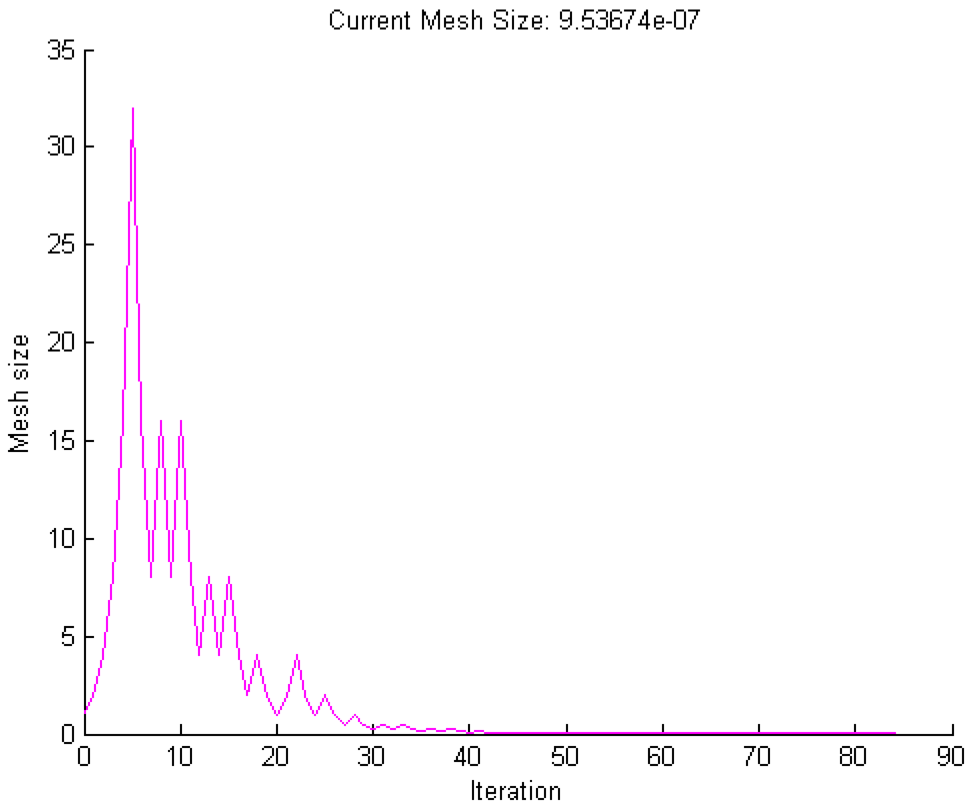


Level of display: iterative

3 Run the optimization.


Run solver and view results
Use random states from previous run
Start
Pause
Stop
Current iteration:
Clear Results

The Optimization app displays the following plot.



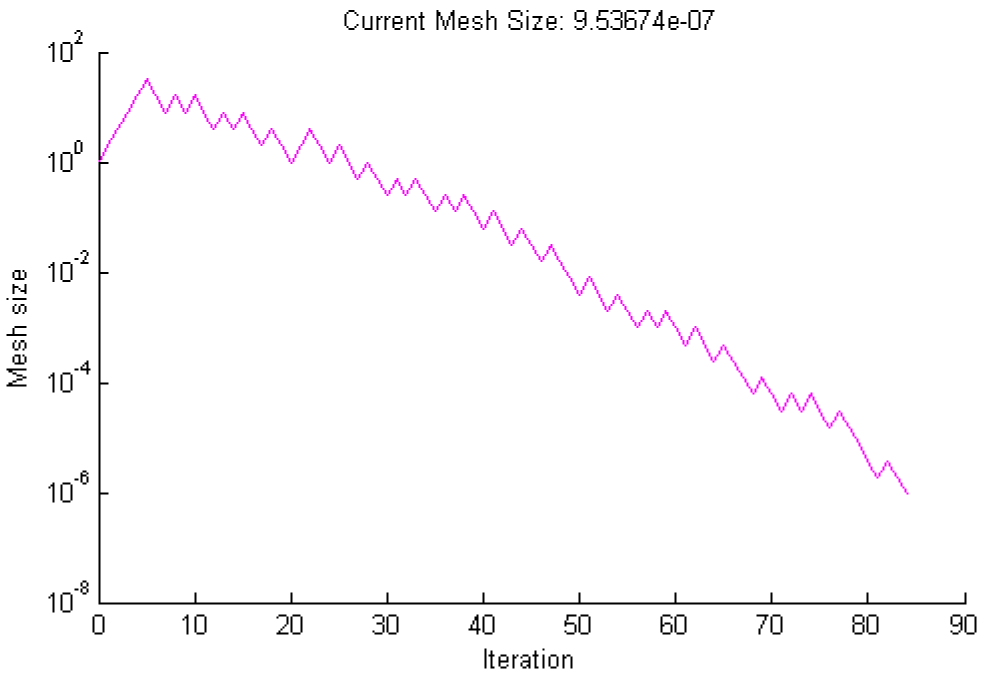
To see the changes in mesh size more clearly, change the *y*-axis to logarithmic scaling as follows:

1 Select **Axes Properties** from the **Edit** menu in the plot window.

2 In the Properties Editor, select the **Y Axis** tab.

3 Set **Scale** to **Log**.

Updating these settings in the MATLAB Property Editor shows the plot in the following figure.



The first 5 iterations result in successful polls, so the mesh sizes increase steadily during this time. You can see that the first unsuccessful poll occurs at iteration 6 by looking at the command-line display:

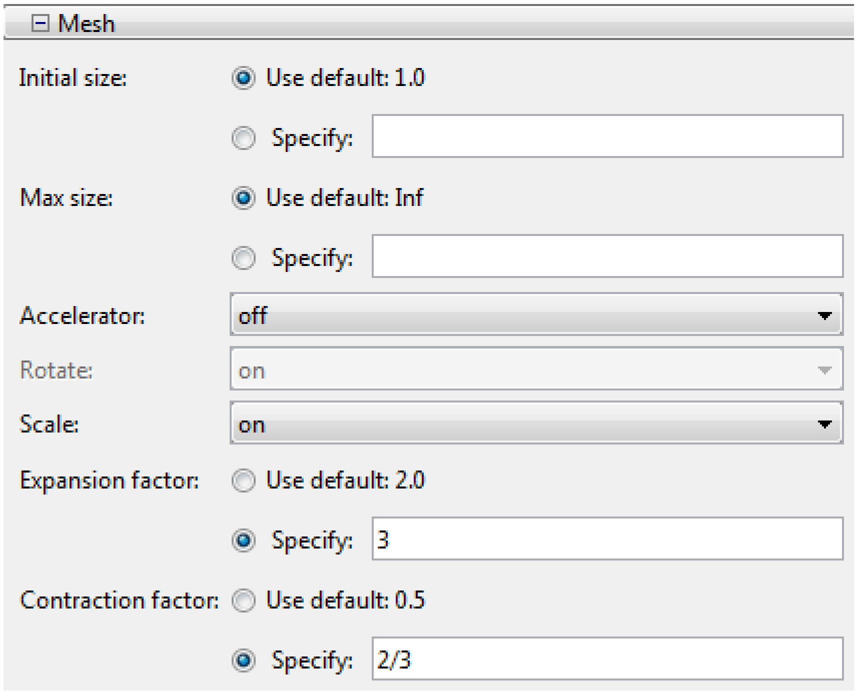
```
Iter     f-count          f(x)      MeshSize      Method
    0         1       2273.76             1
    1         2       2251.69             2     Successful Poll
    2         3       2209.86             4     Successful Poll
    3         4       2135.43             8     Successful Poll
    4         5       2023.48            16     Successful Poll
    5         6       1947.23            32     Successful Poll
    6        15       1947.23            16     Refine Mesh
```

Note that at iteration 5, which is successful, the mesh size doubles for the next iteration. But at iteration 6, which is unsuccessful, the mesh size is multiplied `0.5`.

To see how **Expansion factor** and **Contraction factor** affect the pattern search, make the following changes:

- Set **Expansion factor** to `3.0`.

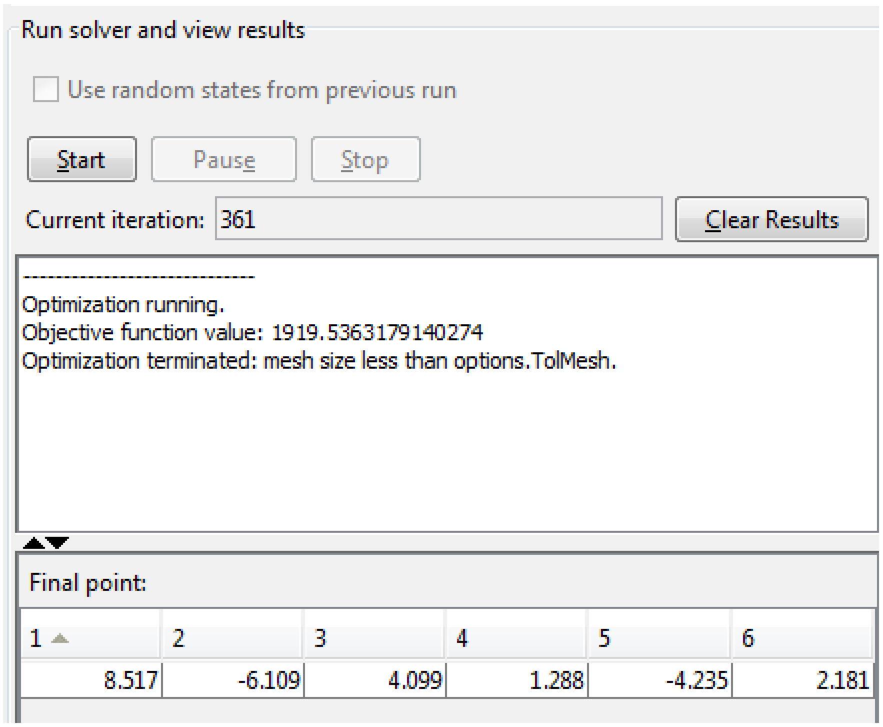
- Set **Contraction factor** to `2/3`.



Mesh

Initial size:	[✓ Use default: 1.0]	
	Specify:	
Max size:	[✓ Use default: Inf]	
	Specify:	
Accelerator:	off	
Rotate:	on	
Scale:	on	
Expansion factor:	Use default: 2.0	
	[✓ Specify:]	3
Contraction factor:	Use default: 0.5	
	[✓ Specify:]	2/3

Then click **Start**. The **Run solver and view results** pane shows that the final point is approximately the same as with the default settings of **Expansion factor** and **Contraction factor**, but that the pattern search takes longer to reach that point.


Run solver and view results
Use random states from previous run
Start
Pause
Stop
Clear Results

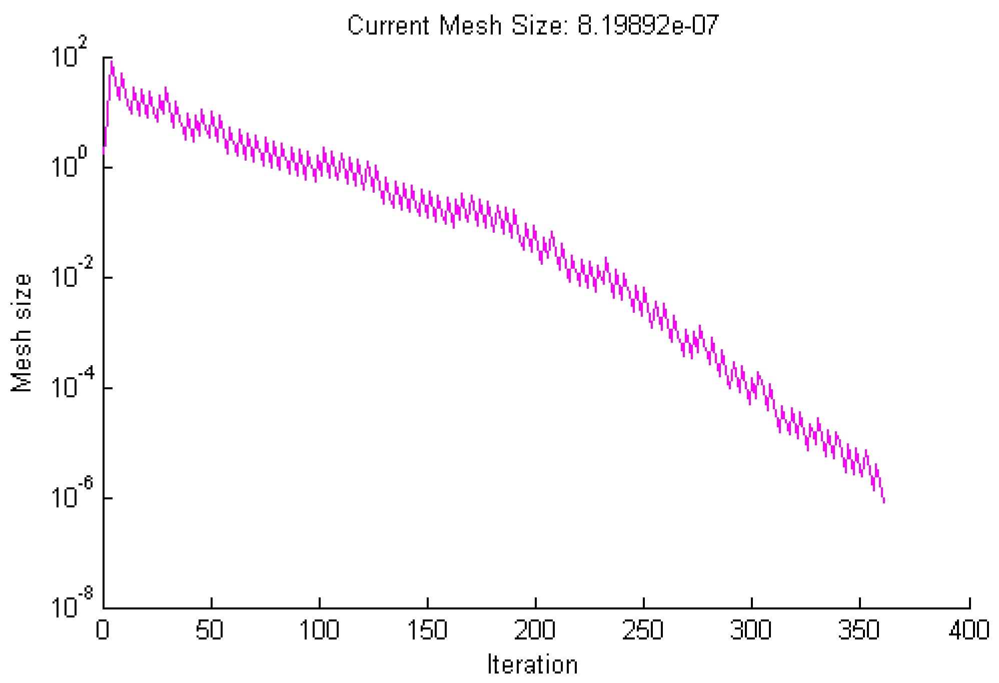
Current iteration: 361

```
-----------------------------
Optimization running.
Objective function value: 1919.5363179140274
Optimization terminated: mesh size less than options.TolMesh.
```

Final point:

1	2	3	4	5	6
8.517	-6.109	4.099	1.288	-4.235	2.181

When you change the scaling of the *y*-axis to logarithmic, the mesh size plot appears as shown in the following figure.



Note that the mesh size increases faster with **Expansion factor** set to `3.0`, as compared with the default value of `2.0`, and decreases more slowly with **Contraction factor** set to `0.75`, as compared with the default value of `0.5`.

Mesh Accelerator

The mesh accelerator can make a pattern search converge faster to an optimal point by reducing the number of iterations required to reach the mesh tolerance. When the mesh size is below a certain value, the pattern search contracts the mesh size by a factor smaller than the **Contraction factor** factor. Mesh accelerator applies only to the GPS and GSS algorithms.

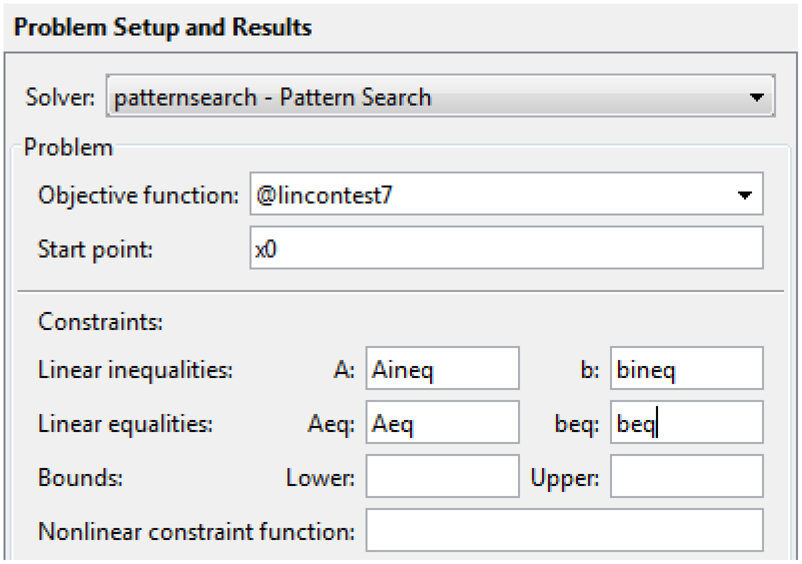
Note For best results, use the mesh accelerator for problems in which the objective function is not too steep near the optimal point, or you might lose some accuracy. For differentiable problems, this means that the absolute value of the derivative is not too large near the solution.

To use the mesh accelerator, set **Accelerator** to `On` in the **Mesh** options. Or, at the command line, set the `MeshAccelerator` option to `'on'`.

For example, set up the problem described in "Linearly Constrained Problem" on page 4-69 as follows:

1. Enter the following at the command line:

   ```
   x0 = [2 1 0 9 1 0];
   Aineq = [-8 7 3 -4 9 0];
   bineq = 7;
   Aeq = [7 1 8 3 3 3; 5 0 -5 1 -5 8; -2 -6 7 1 1 9; 1 -1 2 -2 3 -3];
   beq = [84 62 65 1];
   ```
2. Set up your problem in the Optimization app to match the following figures.



Problem Setup and Results

Solver: patternsearch - Pattern Search

Problem

Objective function: @lincontest7

Start point: x0

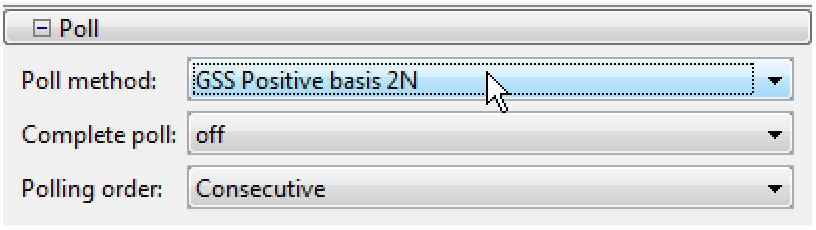
Constraints:

Linear inequalities: A: Aineq b: bineq

Linear equalities: Aeq: Aeq beq: beq

Bounds: Lower: Upper:

Nonlinear constraint function:

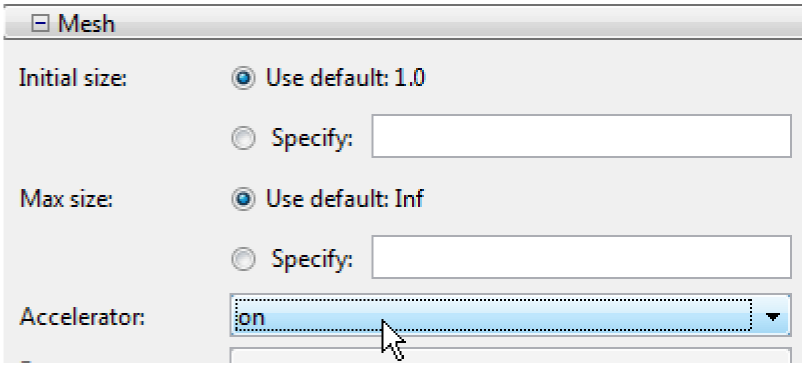


Poll

Poll method: GSS Positive basis 2N

Complete poll: off

Polling order: Consecutive

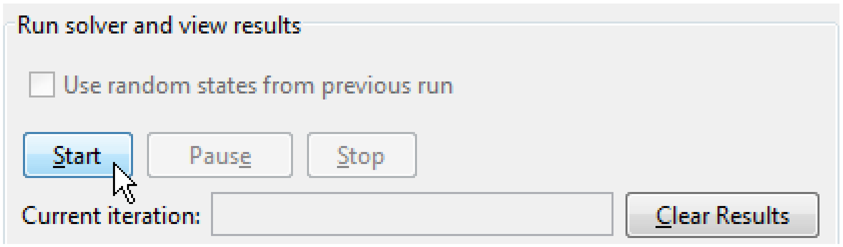


Initial size: ● Use default: 1.0 ○ Specify:

Max size: ● Use default: Inf ○ Specify:

Accelerator: on

3 Run the optimization.


Run solver and view results
Use random states from previous run
Start
Pause
Stop
Current iteration:
Clear Results

The number of iterations required to reach the mesh tolerance is 78, as compared with 84 when **Accelerator** is set to `Off`.

You can see the effect of the mesh accelerator by setting **Level of display** to `Iterative` in **Display to command window**. Run the example with **Accelerator** set to `On`, and then run it again with **Accelerator** set to `Off`. The mesh sizes are the same until iteration 70, but differ at iteration 71. The MATLAB Command Window displays the following lines for iterations 70 and 71 with **Accelerator** set to `Off`.

```
Iter     f-count          f(x)       MeshSize      Method
   70      618         1919.54     6.104e-05     Refine Mesh
   71      630         1919.54     3.052e-05     Refine Mesh
```

Note that the mesh size is multiplied by `0.5`, the default value of **Contraction factor**.

For comparison, the Command Window displays the following lines for the same iteration numbers with **Accelerator** set to `On`.

```
Iter     f-count          f(x)       MeshSize      Method
```

```
   70       618          1919.54     6.104e-05     Refine Mesh
   71       630          1919.54     1.526e-05     Refine Mesh
```

In this case the mesh size is multiplied by `0.25`.

Constrained Minimization Using patternsearch

In this section...
"Linearly Constrained Problem" on page 4-69
"Nonlinearly Constrained Problem" on page 4-72

Linearly Constrained Problem

Problem Description

This section presents an example of performing a pattern search on a constrained minimization problem. The example minimizes the function

$$F(x) = \frac{1}{2} x^T H x + f^T x,$$

where

$$H = \begin{bmatrix} 36 & 17 & 19 & 12 & 8 & 15 \\ 17 & 33 & 18 & 11 & 7 & 14 \\ 19 & 18 & 43 & 13 & 8 & 16 \\ 12 & 11 & 13 & 18 & 6 & 11 \\ 8 & 7 & 8 & 6 & 9 & 8 \\ 15 & 14 & 16 & 11 & 8 & 29 \end{bmatrix},$$
$$f = [20 \quad 15 \quad 21 \quad 18 \quad 29 \quad 24],$$

subject to the constraints

$$A \cdot x \le b,$$
$$Aeq \cdot x = beq,$$

where

$$\begin{aligned}
A &= [-8 \quad 7 \quad 3 \quad -4 \quad 9 \quad 0],\\
b &= 7,\\
Aeq &= \begin{bmatrix} 7 & 1 & 8 & 3 & 3 & 3 \\ 5 & 0 & -5 & 1 & -5 & 8 \\ -2 & -6 & 7 & 1 & 1 & 9 \\ 1 & -1 & 2 & -2 & 3 & -3 \end{bmatrix},\\
beq &= [84 \quad 62 \quad 65 \quad 1].
\end{aligned}$$

Performing a Pattern Search on the Example

To perform a pattern search on the example, first enter

```
optimtool('patternsearch')
```

to open the Optimization app, or enter `optimtool` and then choose `patternsearch` from the **Solver** menu. Then type the following function in the **Objective function** field:

```
@lincontest7
```

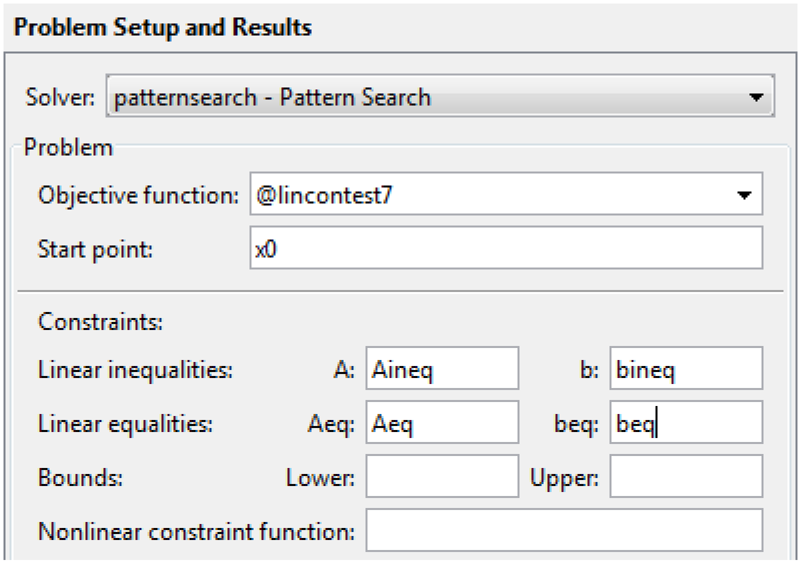
`lincontest7` is a file included in Global Optimization Toolbox software that computes the objective function for the example. Because the matrices and vectors defining the starting point and constraints are large, it is more convenient to set their values as variables in the MATLAB workspace first and then enter the variable names in the Optimization app. To do so, enter

```
x0 = [2 1 0 9 1 0];
Aineq = [-8 7 3 -4 9 0];
bineq = 7;
Aeq = [7 1 8 3 3 3; 5 0 -5 1 -5 8; -2 -6 7 1 1 9; 1 -1 2 -2 3 -3];
beq = [84 62 65 1];
```

Then, enter the following in the Optimization app:

- Set **Start point** to `x0`.
- Set the following **Linear inequalities**:
    - Set **A** to `Aineq`.
    - Set **b** to `bineq`.
    - Set **Aeq** to `Aeq`.
    - Set **beq** to `beq`.

The following figure shows these settings in the Optimization app.



Problem Setup and Results

Solver: patternsearch - Pattern Search

Problem

Objective function: @lincontest7

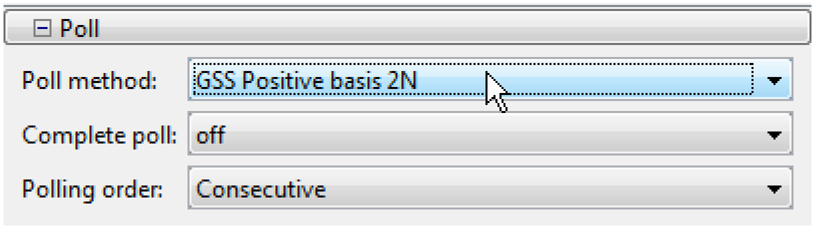
Start point: x0

Constraints:

Linear inequalities:	A:	Aineq	b:	bineq
Linear equalities:	Aeq:	Aeq	beq:	beq
Bounds:	Lower:		Upper:	

Nonlinear constraint function:

Since this is a linearly constrained problem, set the **Poll method** to `GSS Positive basis 2N`. For more information about the efficiency of the GSS search methods for linearly constrained problems, see "Compare the Efficiency of Poll Options" on page 4-51.



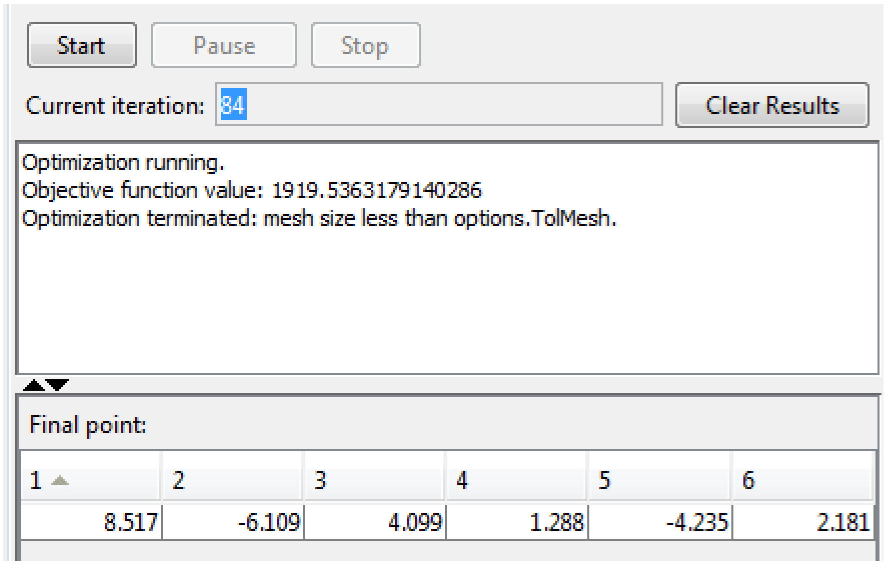
Poll

Poll method: GSS Positive basis 2N

Complete poll: off

Polling order: Consecutive

Then click **Start** to run the pattern search. When the search is finished, the results are displayed in **Run solver and view results** pane, as shown in the following figure.



84

```
Optimization running.
Objective function value: 1919.5363179140286
Optimization terminated: mesh size less than options.TolMesh.
```

Final point:

1	2	3	4	5	6
8.517	-6.109	4.099	1.288	-4.235	2.181

Nonlinearly Constrained Problem

Suppose you want to minimize the simple objective function of two variables `x1` and `x2`,

$$\min_x f(x) = \left(4 - 2.1x_1^2 - x_1^{4/3}\right)x_1^2 + x_1x_2 + \left(-4 + 4x_2^2\right)x_2^2$$

subject to the following nonlinear inequality constraints and bounds

$$\begin{aligned} &x_1x_2 + x_1 - x_2 + 1.5 \le 0 && \text{(nonlinear constraint)} \\ &10 - x_1x_2 \le 0 && \text{(nonlinear constraint)} \\ &0 \le x_1 \le 1 && \text{(bound)} \\ &0 \le x_2 \le 13 && \text{(bound)} \end{aligned}$$

Begin by creating the objective and constraint functions. First, create a file named `simple_objective.m` as follows:

```
function y = simple_objective(x)
y = (4 - 2.1*x(1)^2 + x(1)^4/3)*x(1)^2 + x(1)*x(2) + (-4 + 4*x(2)^2)*x(2)^2;
```

The pattern search solver assumes the objective function will take one input `x` where `x` has as many elements as number of variables in the problem. The objective function computes the value of the function and returns that scalar value in its one return argument `y`.

Then create a file named `simple_constraint.m` containing the constraints:

```
function [c, ceq] = simple_constraint(x)
c = [1.5 + x(1)*x(2) + x(1) - x(2);
-x(1)*x(2) + 10];
ceq = [];
```

The pattern search solver assumes the constraint function will take one input `x`, where `x` has as many elements as the number of variables in the problem. The constraint function computes the values of all the inequality and equality constraints and returns two vectors, `c` and `ceq`, respectively.

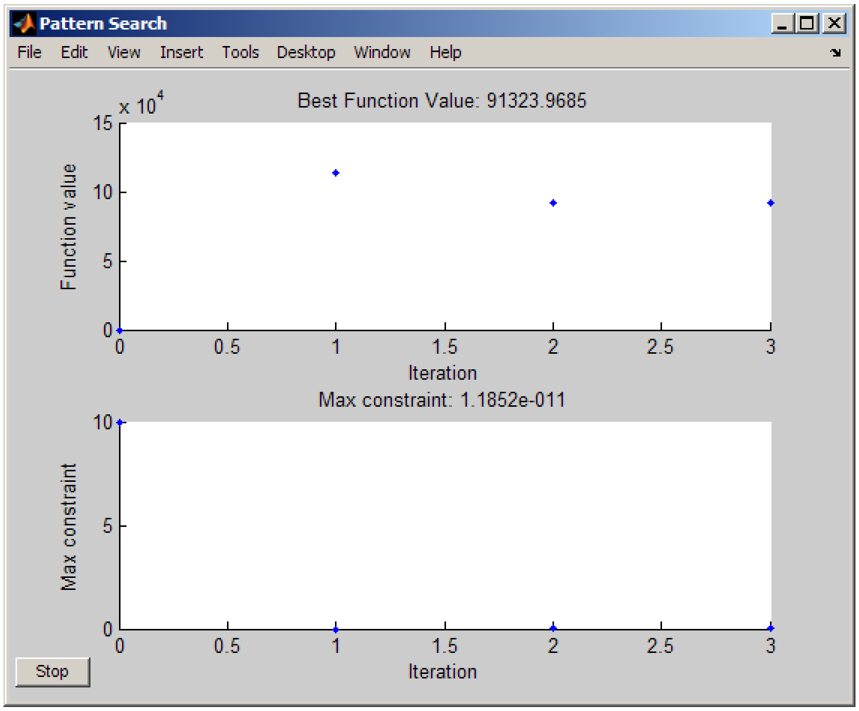
Next, to minimize the objective function using the `patternsearch` function, you need to pass in a function handle to the objective function as well as specifying a start point as the second argument. Lower and upper bounds are provided as `LB` and `UB` respectively. In addition, you also need to pass a function handle to the nonlinear constraint function.

```
ObjectiveFunction = @simple_objective;
X0 = [0 0];   % Starting point
LB = [0 0];   % Lower bound
UB = [1 13];  % Upper bound
ConstraintFunction = @simple_constraint;
[x,fval] = patternsearch(ObjectiveFunction,X0,[],[],[],[],...
                         LB,UB,ConstraintFunction)

Optimization terminated: mesh size less than options.TolMesh
 and constraint violation is less than options.TolCon.

x =
    0.8122   12.3122

fval =
  9.1324e+004
```

Next, plot the results. Create an options structure using `psoptimset` that selects two plot functions. The first plot function `psplotbestf` plots the best objective function value at every iteration. The second plot function `psplotmaxconstr` plots the maximum constraint violation at every iteration.

Note: You can also visualize the progress of the algorithm by displaying information to the Command Window using the `'Display'` option.

```
options = psoptimset('PlotFcns',{@psplotbestf,@psplotmaxconstr},'Display','iter');
[x,fval] = patternsearch(ObjectiveFunction,X0,[],[],[],[],LB,UB,ConstraintFunction,options)

                                       max
  Iter   f-count      f(x)      constraint   MeshSize      Method
    0       1           0            10       0.8919
    1      28      113580             0        0.001   Increase penalty
    2     105       91324     1.788e-07        1e-05   Increase penalty
    3     192       91324     1.187e-11        1e-07   Increase penalty
Optimization terminated: mesh size less than options.TolMesh
 and constraint violation is less than options.TolCon.

x =
    0.8122   12.3122


fval =
  9.1324e+004
```


Pattern Search
File Edit View Insert Tools Desktop Window Help
Best Function Value: 91323.9685
x 10^4
Function value
Iteration
Max constraint: 1.1852e-011
Max constraint
Iteration
Stop

Best Objective Function Value and Maximum Constraint Violation at Each Iteration

Use Cache

Typically, at any given iteration of a pattern search, some of the mesh points might coincide with mesh points at previous iterations. By default, the pattern search recomputes the objective function at these mesh points even though it has already computed their values and found that they are not optimal. If computing the objective function takes a long time—say, several minutes—this can make the pattern search run significantly longer.

You can eliminate these redundant computations by using a cache, that is, by storing a history of the points that the pattern search has already visited. To do so, set **Cache** to `On` in **Cache** options. At each poll, the pattern search checks to see whether the current mesh point is within a specified tolerance, **Tolerance**, of a point in the cache. If so, the search does not compute the objective function for that point, but uses the cached function value and moves on to the next point.

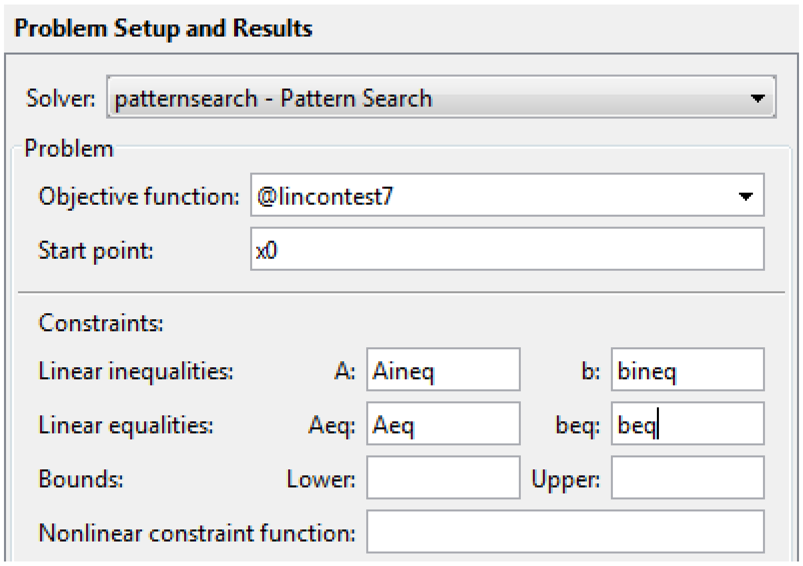
Note When **Cache** is set to `On`, the pattern search might fail to identify a point in the current mesh that improves the objective function because it is within the specified tolerance of a point in the cache. As a result, the pattern search might run for more iterations with **Cache** set to `On` than with **Cache** set to `Off`. It is generally a good idea to keep the value of **Tolerance** very small, especially for highly nonlinear objective functions.

For example, set up the problem described in "Linearly Constrained Problem" on page 4-69 as follows:

1 Enter the following at the command line:

```
x0 = [2 1 0 9 1 0];
Aineq = [-8 7 3 -4 9 0];
bineq = 7;
Aeq = [7 1 8 3 3 3; 5 0 -5 1 -5 8; -2 -6 7 1 1 9; 1 -1 2 -2 3 -3];
beq = [84 62 65 1];
```

2 Set up your problem in the Optimization app to match the following figures.



Problem Setup and Results

Solver: patternsearch - Pattern Search

Problem

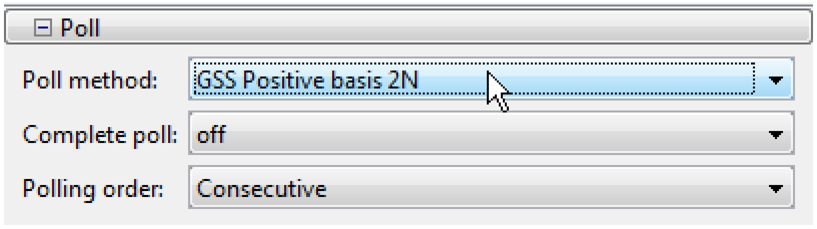
Objective function: @lincontest7

Start point: x0

Constraints:

Linear inequalities:	A:	Aineq	b:	bineq
Linear equalities:	Aeq:	Aeq	beq:	beq
Bounds:	Lower:		Upper:	

Nonlinear constraint function:

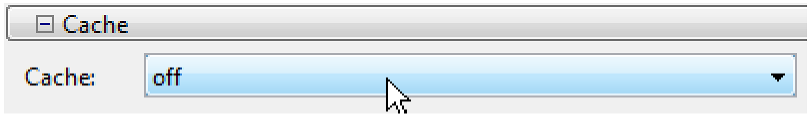


Poll

Poll method: GSS Positive basis 2N

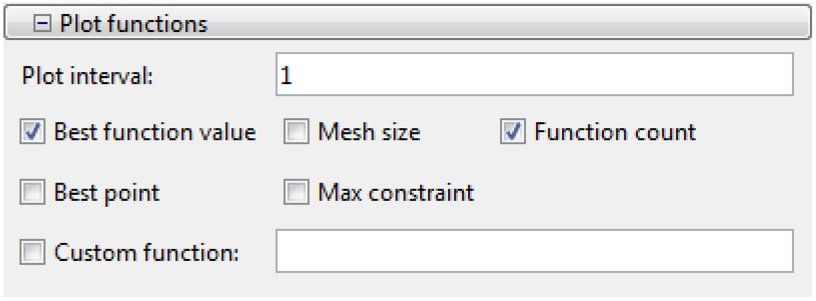
Complete poll: off

Polling order: Consecutive



Cache

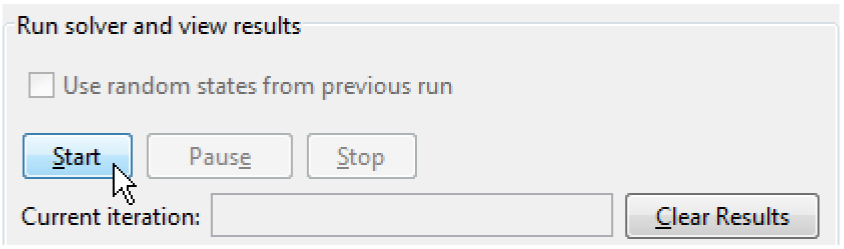
Cache: off



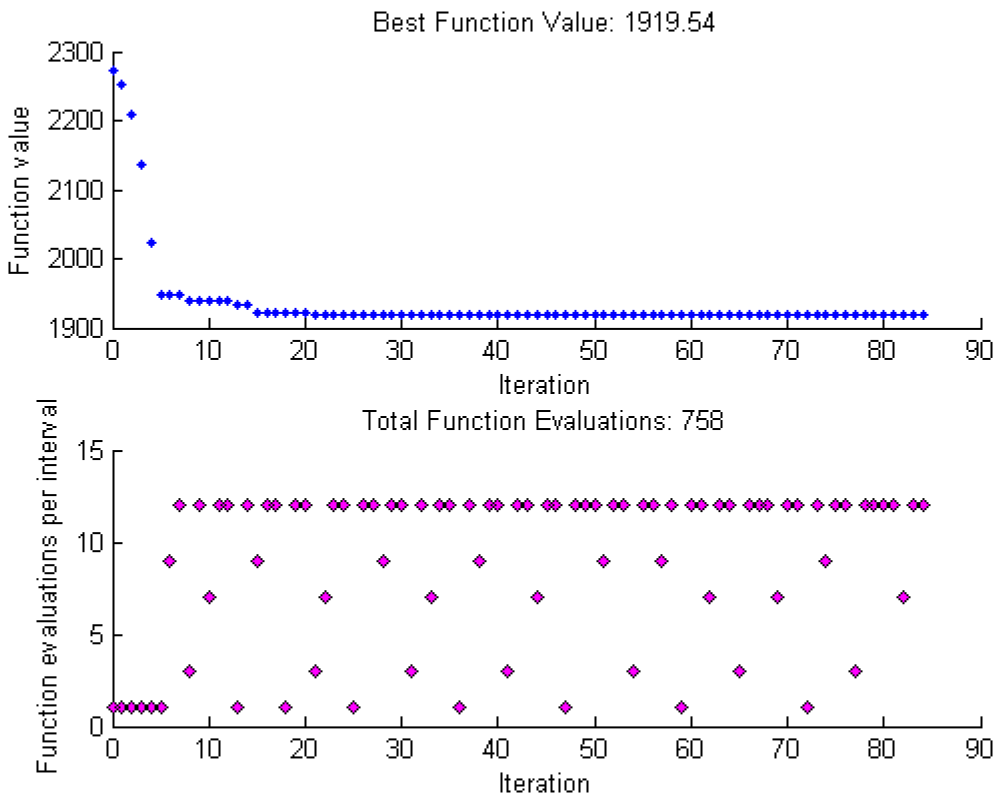
Plot interval: 1

[✓] Best function value
[ ] Mesh size
[✓] Function count
[ ] Best point
[ ] Max constraint
[ ] Custom function:

3 Run the optimization.

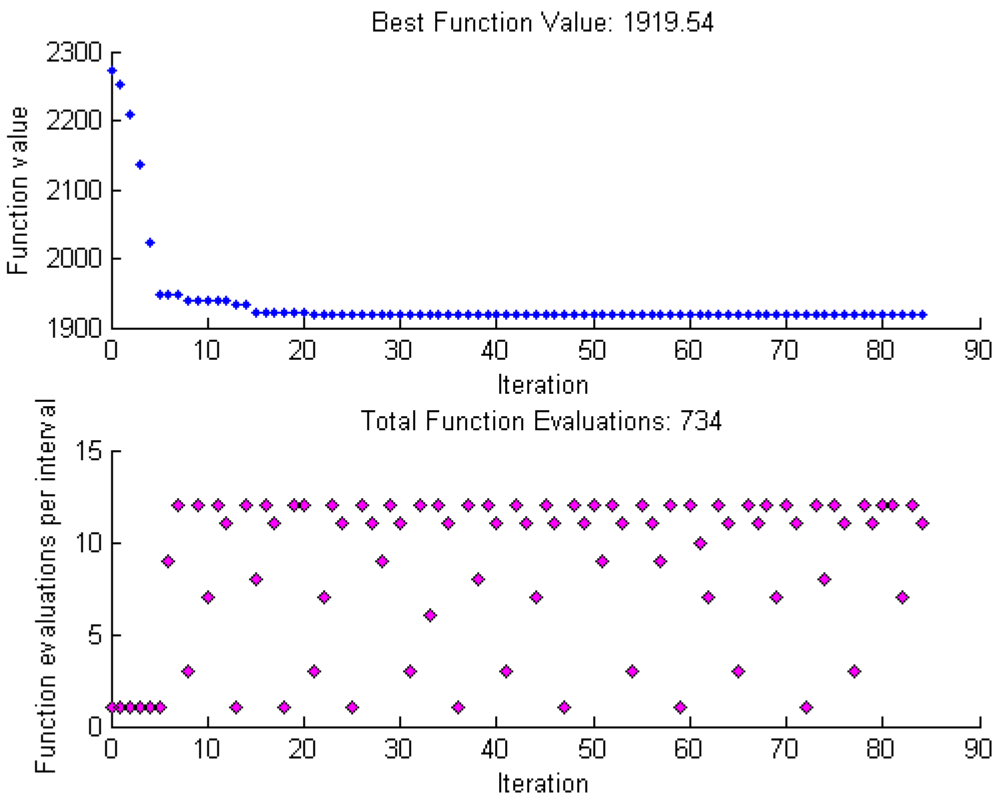

Run solver and view results
Use random states from previous run
Start
Pause
Stop
Current iteration:
Clear Results

After the pattern search finishes, the plots appear as shown in the following figure.



Note that the total function count is 758.

Now, set **Cache** to `On` and run the example again. This time, the plots appear as shown in the following figure.



This time, the total function count is reduced to 734.

Vectorize the Objective and Constraint Functions

In this section...

Vectorize for Speed

Direct search often runs faster if you *vectorize* the objective and nonlinear constraint functions. This means your functions evaluate all the points in a poll or search pattern at once, with one function call, without having to loop through the points one at a time. Therefore, the option `Vectorized` = `'on'` works only when `CompletePoll` or `CompleteSearch` is also set to `'on'`. However, when you set `Vectorized` = `'on'`, `patternsearch` checks that the objective and any nonlinear constraint functions give outputs of the correct shape for vectorized calculations, regardless of the setting of the `CompletePoll` or `CompleteSearch` options.

If there are nonlinear constraints, the objective function and the nonlinear constraints all need to be vectorized in order for the algorithm to compute in a vectorized manner.

Note: Write your vectorized objective function or nonlinear constraint function to accept a matrix with an arbitrary number of points. `patternsearch` sometimes evaluates a single point even during a vectorized calculation.

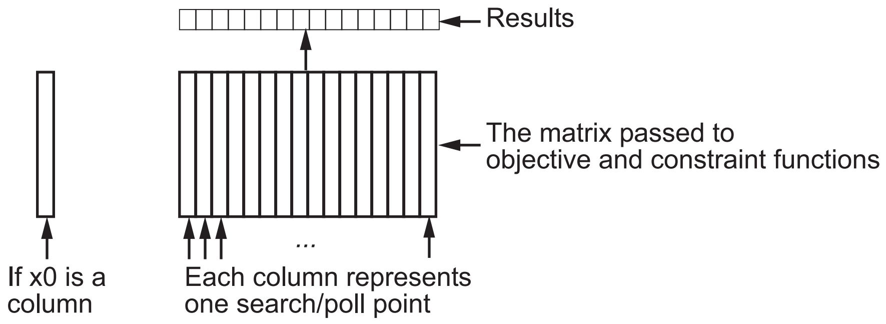
Vectorized Objective Function

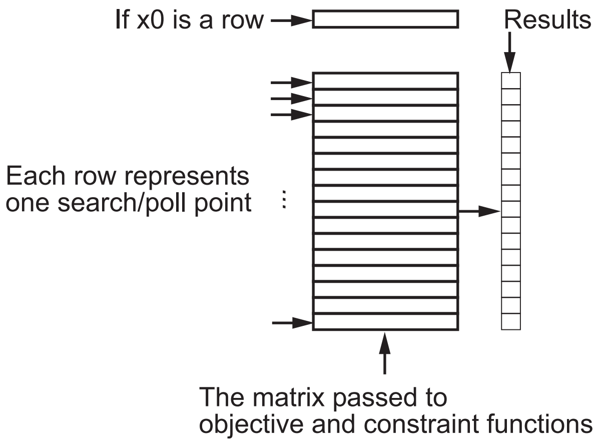
A vectorized objective function accepts a matrix as input and generates a vector of function values, where each function value corresponds to one row or column of the input matrix. `patternsearch` resolves the ambiguity in whether the rows or columns of the matrix represent the points of a pattern as follows. Suppose the input matrix has `m` rows and `n` columns:

- If the initial point `x0` is a column vector of size `m`, the objective function takes each column of the matrix as a point in the pattern and returns a row vector of size `n`.

- If the initial point `x0` is a row vector of size `n`, the objective function takes each row of the matrix as a point in the pattern and returns a column vector of size `m`.
- If the initial point `x0` is a scalar, `patternsearch` assumes that `x0` is a row vector. Therefore, the input matrix has one column (`n` = 1, the input matrix is a vector), and each entry of the matrix represents one row for the objective function to evaluate. The output of the objective function in this case is a column vector of size `m`.

Pictorially, the matrix and calculation are represented by the following figure.


Results
The matrix passed to objective and constraint functions
...
If x0 is a column
Each column represents one search/poll point


If x0 is a row
Results
Each row represents one search/poll point
⋮
The matrix passed to objective and constraint functions

Structure of Vectorized Functions

For example, suppose the objective function is

$$f(x) = x_1^4 + x_2^4 - 4x_1^2 - 2x_2^2 + 3x_1 - x_2 / 2.$$

If the initial vector `x0` is a column vector, such as `[0;0]`, a function for vectorized evaluation is

```
function f = vectorizedc(x)

f = x(1,:).^4+x(2,:).^4-4*x(1,:).^2-2*x(2,:).^2 ...
    +3*x(1,:)-.5*x(2,:);
```

If the initial vector `x0` is a row vector, such as `[0,0]`, a function for vectorized evaluation is

```
function f = vectorizedr(x)

f = x(:,1).^4+x(:,2).^4-4*x(:,1).^2-2*x(:,2).^2 ...
    +3*x(:,1)-.5*x(:,2);
```

Tip If you want to use the same objective (fitness) function for both pattern search and genetic algorithm, write your function to have the points represented by row vectors, and write `x0` as a row vector. The genetic algorithm always takes individuals as the rows of a matrix. This was a design decision—the genetic algorithm does not require a user-supplied population, so needs to have a default format.

To minimize `vectorizedc`, enter the following commands:

```
options=psoptimset('Vectorized','on','CompletePoll','on');
x0=[0;0];
[x fval]=patternsearch(@vectorizedc,x0,...
         [],[],[],[],[],[],[],options)
```

MATLAB returns the following output:

```
Optimization terminated: mesh size less than options.TolMesh.

x =
   -1.5737
    1.0575

fval =
  -10.0088
```

Vectorized Constraint Functions

Only nonlinear constraints need to be vectorized; bounds and linear constraints are handled automatically. If there are nonlinear constraints, the objective function and the nonlinear constraints all need to be vectorized in order for the algorithm to compute in a vectorized manner.

The same considerations hold for constraint functions as for objective functions: the initial point `x0` determines the type of points (row or column vectors) in the poll or search. If the initial point is a row vector of size k, the matrix x passed to the constraint function has k columns. Similarly, if the initial point is a column vector of size k, the matrix of poll or search points has k rows. The figure Structure of Vectorized Functions may make this clear. If the initial point is a scalar, `patternsearch` assumes that it is a row vector.

Your nonlinear constraint function returns two matrices, one for inequality constraints, and one for equality constraints. Suppose there are n_c nonlinear inequality constraints and n_{ceq} nonlinear equality constraints. For row vector `x0`, the constraint matrices have n_c and n_{ceq} columns respectively, and the number of rows is the same as in the input matrix. Similarly, for a column vector `x0`, the constraint matrices have n_c and n_{ceq} rows respectively, and the number of columns is the same as in the input matrix. In figure Structure of Vectorized Functions, “Results” includes both n_c and n_{ceq}.

Example of Vectorized Objective and Constraints

Suppose that the nonlinear constraints are

$$\frac{x_1^2}{9}+\frac{x_2^2}{4}\le 1 \text{ (the interior of an ellipse),}$$
$$x_2 \ge \cosh(x_1)-1.$$

Write a function for these constraints for row-form `x0` as follows:

```
function [c ceq] = ellipsecosh(x)

c(:,1)=x(:,1).^2/9+x(:,2).^2/4-1;
c(:,2)=cosh(x(:,1))-x(:,2)-1;
ceq=[];
```

Minimize `vectorizedr` (defined in "Vectorized Objective Function" on page 4-80) subject to the constraints `ellipsecosh`:

```
x0=[0,0];
options=psoptimset('Vectorized','on','CompletePoll','on');
[x fval]=patternsearch(@vectorizedr,x0,...
         [],[],[],[],[],[],@ellipsecosh,options)
```

MATLAB returns the following output:

```
Optimization terminated: mesh size less than options.TolMesh
 and constraint violation is less than options.TolCon.

x =
   -1.3516    1.0612

fval =
   -9.5394
```

Optimize an ODE in Parallel

This example shows how to optimize parameters of an ODE.

It also shows how to avoid computing the objective and nonlinear constraint function twice when the ODE solution returns both. The example compares `patternsearch` and `ga` in terms of time to run the solver and the quality of the solutions.

You need a Parallel Computing Toolbox license to use parallel computing.

Step 1. Define the problem.

The problem is to change the position and angle of a cannon to fire a projectile as far as possible beyond a wall. The cannon has a muzzle velocity of 300 m/s. The wall is 20 m high. If the cannon is too close to the wall, it has to fire at too steep an angle, and the projectile does not travel far enough. If the cannon is too far from the wall, the projectile does not travel far enough either.

Air resistance slows the projectile. The resisting force is proportional to the square of the velocity, with proportionality constant 0.02. Gravity acts on the projectile, accelerating it downward with constant 9.81 m/s^2. Therefore, the equations of motion for the trajectory $x(t)$ are

$$\frac{d^2x(t)}{dt^2} = -0.02\left\| x(t) \right\| x(t) - (0, 9.81).$$

The initial position `x0` and initial velocity `xp0` are 2-D vectors. However, the initial height `x0(2)` is 0, so the initial position depends only on the scalar `x0(1)`. And the initial velocity `xp0` has magnitude 300 (the muzzle velocity), so depends only on the initial angle, a scalar. For an initial angle `th`, `xp0 = 300*(cos(th),sin(th))`. Therefore, the optimization problem depends only on two scalars, so it is a 2-D problem. Use the horizontal distance and the angle as the decision variables.

Step 2. Formulate the ODE model.

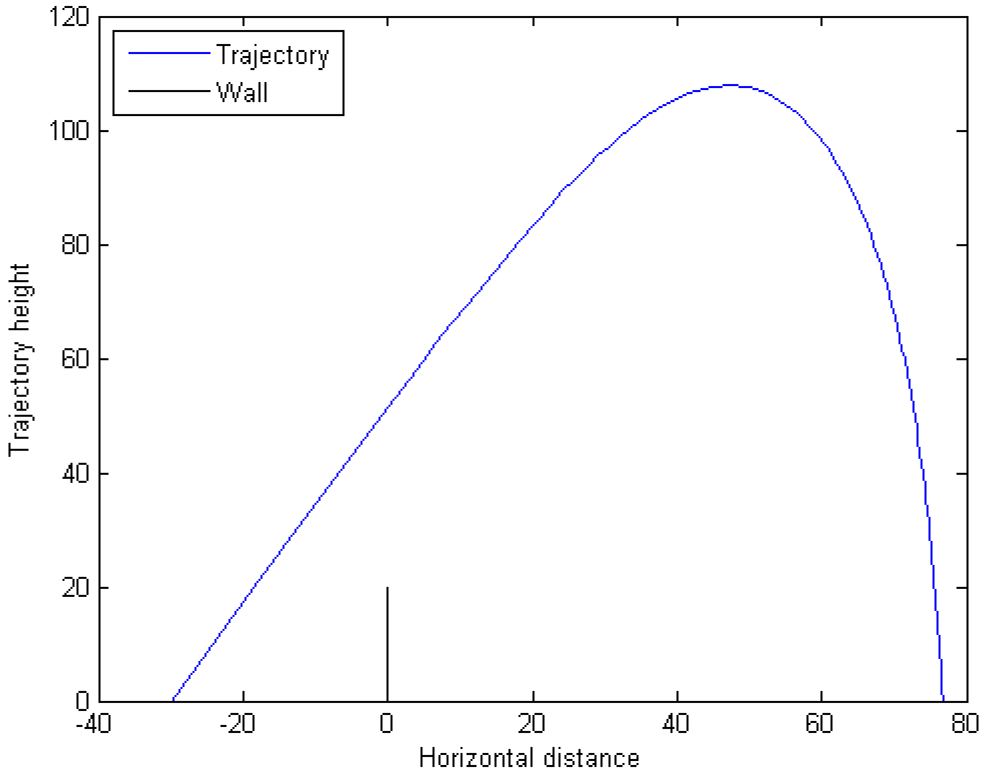
ODE solvers require you to formulate your model as a first-order system. Augment the trajectory vector $(x_1(t),x_2(t))$ with its time derivative $(x'_1(t),x'_2(t))$ to form a 4-D trajectory vector. In terms of this augmented vector, the differential equation becomes

$$\frac{d}{dt}x(t)=\begin{bmatrix} x_3(t) \\ x_4(t) \\ -.02\left\|\left(x_3(t),x_4(t)\right)\right\|x_3(t) \\ -.02\left\|\left(x_3(t),x_4(t)\right)\right\|x_4(t)-9.81 \end{bmatrix}.$$

Write the differential equation as a function file, and save it on your MATLAB path.

```
function f = cannonfodder(t,x)

f = [x(3);x(4);x(3);x(4)]; % initial, gets f(1) and f(2) correct
nrm = norm(x(3:4)) * .02; % norm of the velocity times constant
f(3) = -x(3)*nrm; % horizontal acceleration
f(4) = -x(4)*nrm - 9.81; % vertical acceleration
```

Visualize the solution of the ODE starting 30 m from the wall at an angle of `pi/3`.



Code for generating the figure

```
x0 = [-30;0;300*cos(pi/3);300*sin(pi/3)];
sol = ode45(@cannonfodder,[0,10],x0);
% Find the time when the projectile lands
zerofnd = fzero(@(r)deval(sol,r,2),[sol.x(2),sol.x(end)]);
t = linspace(0,zerofnd); % equal times for plot
xs = deval(sol,t,1); % interpolated x values
ys = deval(sol,t,2); % interpolated y values
plot(xs,ys)
hold on
plot([0,0],[0,20],'k') % Draw the wall
xlabel('Horizontal distance')
ylabel('Trajectory height')
```

```
legend('Trajectory','Wall','Location','NW')
ylim([0 120])
hold off
```

Step 3. Solve using patternsearch.

The problem is to find initial position `x0(1)` and initial angle `x0(2)` to maximize the distance from the wall the projectile lands. Because this is a maximization problem, minimize the negative of the distance (see "Maximizing vs. Minimizing" on page 2-5).

To use `patternsearch` to solve this problem, you must provide the objective, constraint, initial guess, and options.

These two files are the objective and constraint functions. Copy them to a folder on your MATLAB path.

```
function f = cannonobjective(x)
x0 = [x(1);0;300*cos(x(2));300*sin(x(2))];

sol = ode45(@cannonfodder,[0,15],x0);

% Find the time t when y_2(t) = 0
zerofnd = fzero(@(r)deval(sol,r,2),[sol.x(2),sol.x(end)]);
% Then find the x-position at that time
f = deval(sol,zerofnd,1);

f = -f; % take negative of distance for maximization
```

```
function [c,ceq] = cannonconstraint(x)

ceq = [];
x0 = [x(1);0;300*cos(x(2));300*sin(x(2))];

sol = ode45(@cannonfodder,[0,15],x0);

if sol.y(1,end) <= 0 % projectile never reaches wall
    c = 20 - sol.y(2,end);
else
    % Find when the projectile crosses x = 0
    zerofnd = fzero(@(r)deval(sol,r,1),[sol.x(2),sol.x(end)]);
    % Then find the height there, and subtract from 20
    c = 20 - deval(sol,zerofnd,2);
```

```
end
```

Notice that the objective and constraint functions set their input variable `x0` to a 4-D initial point for the ODE solver. The ODE solver does not stop if the projectile hits the wall. Instead, the constraint function simply becomes positive, indicating an infeasible initial value.

The initial position `x0(1)` cannot be above 0, and it is futile to have it be below –200. (It should be near –20 because, with no air resistance, the longest trajectory would start at –20 at an angle `pi/4`.) Similarly, the initial angle `x0(2)` cannot be below 0, and cannot be above `pi/2`. Set bounds slightly away from these initial values:

```
lb = [-200;0.05];
ub = [-1;pi/2-.05];
x0 = [-30,pi/3]; % initial guess
```

Set the `CompletePoll` option to `'on'`. This gives a higher-quality solution, and enables direct comparison with parallel processing, because computing in parallel requires this setting.

```
opts = psoptimset('CompletePoll','on');
```

Call `patternsearch` to solve the problem.

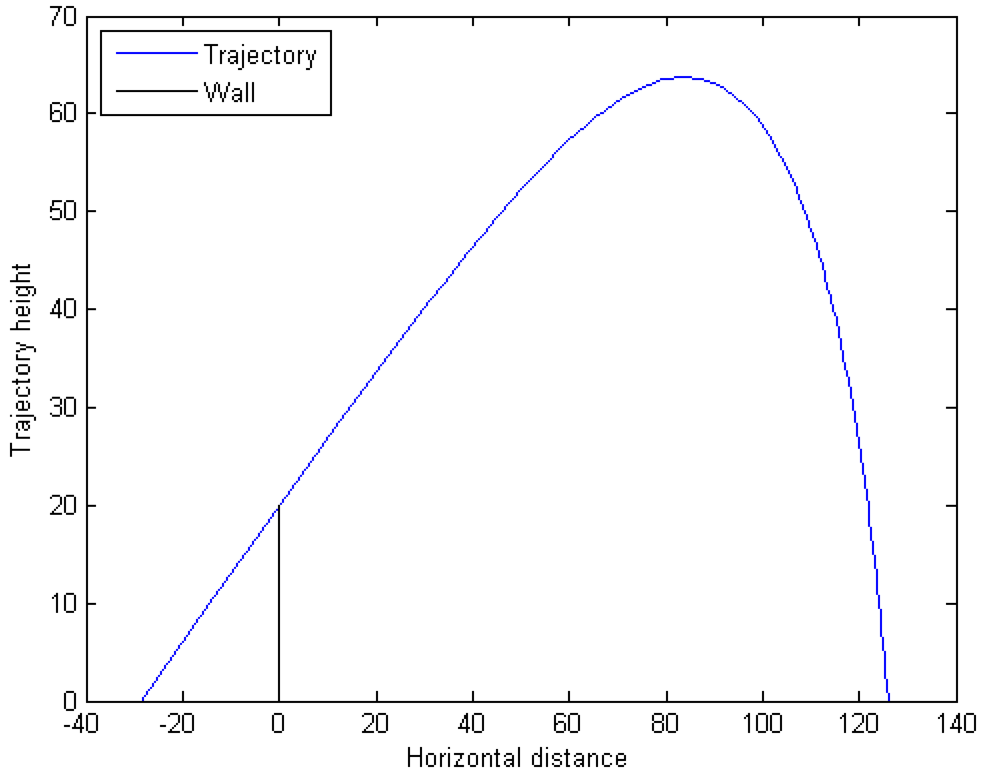
```
tic % time the solution
[xsolution,distance,eflag,outpt] = patternsearch(@cannonobjective,x0,...
    [],[],[],[],lb,ub,@cannonconstraint,opts)
toc

Optimization terminated: mesh size less than options.TolMesh
 and constraint violation is less than options.TolCon.

xsolution =
  -28.8123    0.6095

distance =
 -125.9880

eflag =
     1

outpt =
         function: @cannonobjective
```

```
      problemtype: 'nonlinearconstr'
       pollmethod: 'gpspositivebasis2n'
     searchmethod: []
       iterations: 5
        funccount: 269
         meshsize: 8.9125e-07
    maxconstraint: 0
          message: [1x115 char]

Elapsed time is 3.174088 seconds.
```

Starting the projectile about 29 m from the wall at an angle 0.6095 radian results in the farthest distance, about 126 m. The reported distance is negative because the objective function is the negative of the distance to the wall.

Visualize the solution.

```
x0 = [xsolution(1);0;300*cos(xsolution(2));300*sin(xsolution(2))];

sol = ode45(@cannonfodder,[0,15],x0);
% Find the time when the projectile lands
zerofnd = fzero(@(r)deval(sol,r,2),[sol.x(2),sol.x(end)]);
t = linspace(0,zerofnd); % equal times for plot
xs = deval(sol,t,1); % interpolated x values
ys = deval(sol,t,2); % interpolated y values
plot(xs,ys)
hold on
plot([0,0],[0,20],'k') % Draw the wall
xlabel('Horizontal distance')
ylabel('Trajectory height')
legend('Trajectory','Wall','Location','NW')
ylim([0 70])
hold off
```



Step 4. Avoid calling the expensive subroutine twice.

Both the objective and nonlinear constraint function call the ODE solver to calculate their values. Use the technique in "Objective and Nonlinear Constraints in the Same Function" to avoid calling the solver twice. The `runcannon` file implements this technique. Copy this file to a folder on your MATLAB path.

```
function [x,f,eflag,outpt] = runcannon(x0,opts)

if nargin == 1 % No options supplied
    opts = [];
end

xLast = []; % Last place ode solver was called
```

```
sol = []; % ODE solution structure

fun = @objfun; % the objective function, nested below
cfun = @constr; % the constraint function, nested below

lb = [-200;0.05];
ub = [-1;pi/2-.05];

% Call patternsearch
[x,f,eflag,outpt] = patternsearch(fun,x0,[],[],[],[],lb,ub,cfun,opts);

    function y = objfun(x)
        if ~isequal(x,xLast) % Check if computation is necessary
            x0 = [x(1);0;300*cos(x(2));300*sin(x(2))];
            sol = ode45(@cannonfodder,[0,15],x0);
            xLast = x;
        end
        % Now compute objective function
        % First find when the projectile hits the ground
        zerofnd = fzero(@(r)deval(sol,r,2),[sol.x(2),sol.x(end)]);
        % Then compute the x-position at that time
        y = deval(sol,zerofnd,1);
        y = -y; % take negative of distance
    end

    function [c,ceq] = constr(x)
       ceq = [];
        if ~isequal(x,xLast) % Check if computation is necessary
            x0 = [x(1);0;300*cos(x(2));300*sin(x(2))];
            sol = ode45(@cannonfodder,[0,15],x0);
            xLast = x;
        end
        % Now compute constraint functions
        % First find when the projectile crosses x = 0
        zerofnd = fzero(@(r)deval(sol,r,1),[sol.x(1),sol.x(end)]);
        % Then find the height there, and subtract from 20
        c = 20 - deval(sol,zerofnd,2);
    end

end
```

Reinitialize the problem and time the call to `runcannon`.

```
x0 = [-30;pi/3];
tic
```

```
[xsolution,distance,eflag,outpt] = runcannon(x0,opts);
toc
```

```
Elapsed time is 2.610590 seconds.
```

The solver ran faster than before. If you examine the solution, you see that the output is identical.

Step 5. Compute in parallel.

Try to save more time by computing in parallel. Begin by opening a parallel pool of workers.

```
parpool
```

```
Starting parpool using the 'local' profile ... connected to 4 workers.

ans =

 Pool with properties:

            Connected: true
           NumWorkers: 4
              Cluster: local
        AttachedFiles: {}
          IdleTimeout: 30 minute(s) (30 minutes remaining)
          SpmdEnabled: true
```

Set the options to use parallel computing, and rerun the solver.

```
opts = psoptimset(opts,'UseParallel',true);
x0 = [-30;pi/3];
tic
[xsolution,distance,eflag,outpt] = runcannon(x0,opts);
toc
```

```
Elapsed time is 3.917971 seconds.
```

In this case, parallel computing was slower. If you examine the solution, you see that the output is identical.

Step 6. Compare with the genetic algorithm.

You can also try to solve the problem using the genetic algorithm. However, the genetic algorithm is usually slower and less reliable.

The runcannonga file calls the genetic algorithm and avoids double evaluation of the ODE solver. It resembles runcannon, but calls ga instead of patternsearch, and also checks whether the trajectory reaches the wall. Copy this file to a folder on your MATLAB path.

```
function [x,f,eflag,outpt] = runcannonga(opts)

if nargin == 1 % No options supplied
    opts = [];
end

xLast = []; % Last place ode solver was called
sol = []; % ODE solution structure

fun = @objfun; % the objective function, nested below
cfun = @constr; % the constraint function, nested below

lb = [-200;0.05];
ub = [-1;pi/2-.05];

% Call ga
[x,f,eflag,outpt] = ga(fun,2,[],[],[],[],lb,ub,cfun,opts);

    function y = objfun(x)
        if ~isequal(x,xLast) % Check if computation is necessary
            x0 = [x(1);0;300*cos(x(2));300*sin(x(2))];
            sol = ode45(@cannonfodder,[0,15],x0);
            xLast = x;
        end
        % Now compute objective function
        % First find when the projectile hits the ground
        zerofnd = fzero(@(r)deval(sol,r,2),[sol.x(2),sol.x(end)]);
        % Then compute the x-position at that time
        y = deval(sol,zerofnd,1);
        y = -y; % take negative of distance
    end

    function [c,ceq] = constr(x)
       ceq = [];
        if ~isequal(x,xLast) % Check if computation is necessary
            x0 = [x(1);0;300*cos(x(2));300*sin(x(2))];
            sol = ode45(@cannonfodder,[0,15],x0);
            xLast = x;
        end
        % Now compute constraint functions
        if sol.y(1,end) <= 0 % projectile never reaches wall
            c = 20 - sol.y(2,end);
        else
            % Find when the projectile crosses x = 0
            zerofnd = fzero(@(r)deval(sol,r,1),[sol.x(2),sol.x(end)]);
            % Then find the height there, and subtract from 20
            c = 20 - deval(sol,zerofnd,2);
        end
    end

end
```

Call runcannonga in parallel.

```
opts = gaoptimset('UseParallel',true);
rng default % for reproducibility
tic
[xsolution,distance,eflag,outpt] = runcannonga(opts)
toc

Optimization terminated: average change in the fitness value less than
options.TolFun and constraint violation is less than options.TolCon.

xsolution =
  -17.9172    0.8417

distance =
 -116.6263

eflag =
     1

outpt =
      problemtype: 'nonlinearconstr'
         rngstate: [1x1 struct]
      generations: 5
        funccount: 20212
          message: [1x140 char]
    maxconstraint: 0

Elapsed time is 119.630284 seconds.
```

The `ga` solution is not as good as the `patternsearch` solution: 117 m versus 126 m. `ga` took much more time: about 120 s versus under 5 s.

Related Examples

- "Objective and Nonlinear Constraints in the Same Function"

More About

- "Parallel Computing"

Using the Genetic Algorithm

What Is the Genetic Algorithm?

The genetic algorithm is a method for solving both constrained and unconstrained optimization problems that is based on natural selection, the process that drives biological evolution. The genetic algorithm repeatedly modifies a population of individual solutions. At each step, the genetic algorithm selects individuals at random from the current population to be parents and uses them to produce the children for the next generation. Over successive generations, the population "evolves" toward an optimal solution. You can apply the genetic algorithm to solve a variety of optimization problems that are not well suited for standard optimization algorithms, including problems in which the objective function is discontinuous, nondifferentiable, stochastic, or highly nonlinear. The genetic algorithm can address problems of *mixed integer programming*, where some components are restricted to be integer-valued.

The genetic algorithm uses three main types of rules at each step to create the next generation from the current population:

- *Selection rules* select the individuals, called *parents*, that contribute to the population at the next generation.
- *Crossover rules* combine two parents to form children for the next generation.
- *Mutation rules* apply random changes to individual parents to form children.

The genetic algorithm differs from a classical, derivative-based, optimization algorithm in two main ways, as summarized in the following table.

Classical Algorithm	Genetic Algorithm
Generates a single point at each iteration. The sequence of points approaches an optimal solution.	Generates a population of points at each iteration. The best point in the population approaches an optimal solution.
Selects the next point in the sequence by a deterministic computation.	Selects the next population by computation which uses random number generators.

Optimize Using ga

In this section...

Calling the Function ga at the Command Line

To use the genetic algorithm at the command line, call the genetic algorithm function `ga` with the syntax

```
[x fval] = ga(@fitnessfun, nvars, options)
```

where

- `@fitnessfun` is a handle to the fitness function.
- `nvars` is the number of independent variables for the fitness function.
- `options` is a structure containing options for the genetic algorithm. If you do not pass in this argument, `ga` uses its default options.

The results are given by

- `x` — Point at which the final value is attained
- `fval` — Final value of the fitness function

For an example, see "Finding the Minimum from the Command Line" on page 5-10.

Using the function `ga` is convenient if you want to

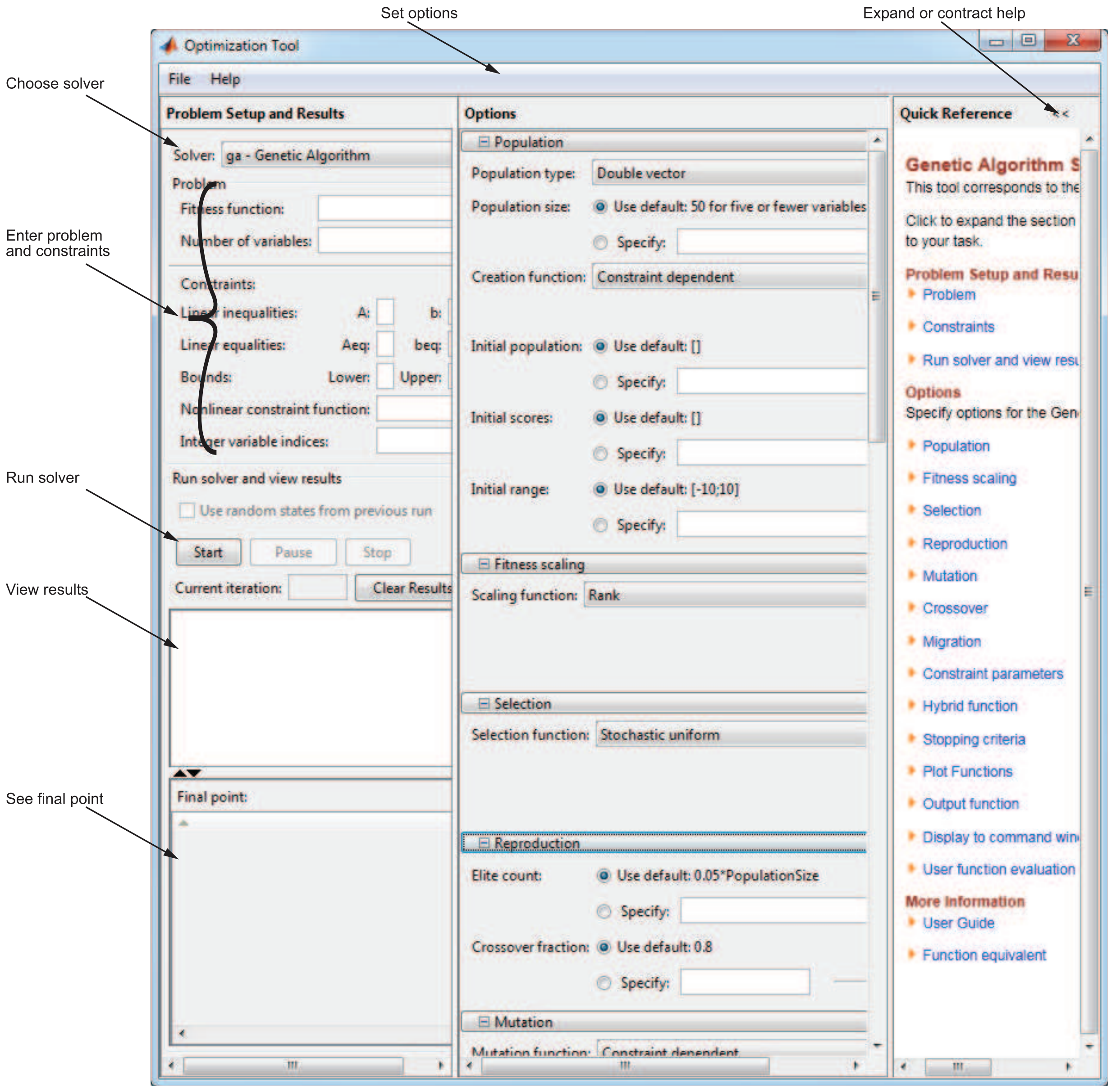
- Return results directly to the MATLAB workspace
- Run the genetic algorithm multiple times with different options, by calling `ga` from a file

Use the Optimization App

To open the Optimization app, enter

```
optimtool('ga')
```

at the command line, or enter `optimtool` and then choose `ga` from the **Solver** menu.



Solver: ga - Genetic Algorithm

Problem
Fitness function:
Number of variables:

Constraints:
Linear inequalities: A: b:
Linear equalities: Aeq: beq:
Bounds: Lower: Upper:
Nonlinear constraint function:
Integer variable indices:

Run solver and view results
Use random states from previous run
Current iteration:

Final point:

Population
Population type: Double vector
Population size: (●) Use default: 50 for five or fewer variables ( ) Specify:
Creation function: Constraint dependent
Initial population: (●) Use default: [] ( ) Specify:
Initial scores: (●) Use default: [] ( ) Specify:
Initial range: (●) Use default: [-10;10] ( ) Specify:

Fitness scaling
Scaling function: Rank

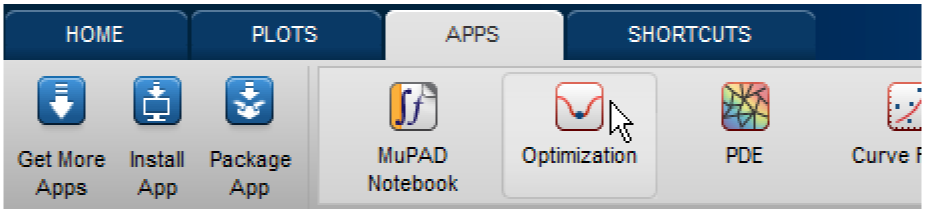
Selection
Selection function: Stochastic uniform

Reproduction
Elite count: (●) Use default: 0.05*PopulationSize ( ) Specify:
Crossover fraction: (●) Use default: 0.8 ( ) Specify:

Mutation
Mutation function: Constraint dependent

Genetic Algorithm S
This tool corresponds to the
Click to expand the section to your task.
Specify options for the Gen

You can also start the tool from the MATLAB **Apps** tab.



To use the Optimization app, you must first enter the following information:

- **Fitness function** — The objective function you want to minimize. Enter the fitness function in the form `@fitnessfun`, where `fitnessfun.m` is a file that computes the fitness function. "Compute Objective Functions" on page 2-2 explains how write this file. The `@` sign creates a function handle to `fitnessfun`.
- **Number of variables** — The length of the input vector to the fitness function. For the function `my_fun` described in "Compute Objective Functions" on page 2-2, you would enter `2`.

You can enter constraints or a nonlinear constraint function for the problem in the **Constraints** pane. If the problem is unconstrained, leave these fields blank.

To run the genetic algorithm, click the **Start** button. The tool displays the results of the optimization in the **Run solver and view results** pane.

You can change the options for the genetic algorithm in the **Options** pane. To view the options in one of the categories listed in the pane, click the + sign next to it.

For more information,

- See "Optimization App" in the Optimization Toolbox documentation.
- See "Minimize Rastrigin's Function" on page 5-6 for an example of using the tool.

Minimize Rastrigin's Function

In this section...

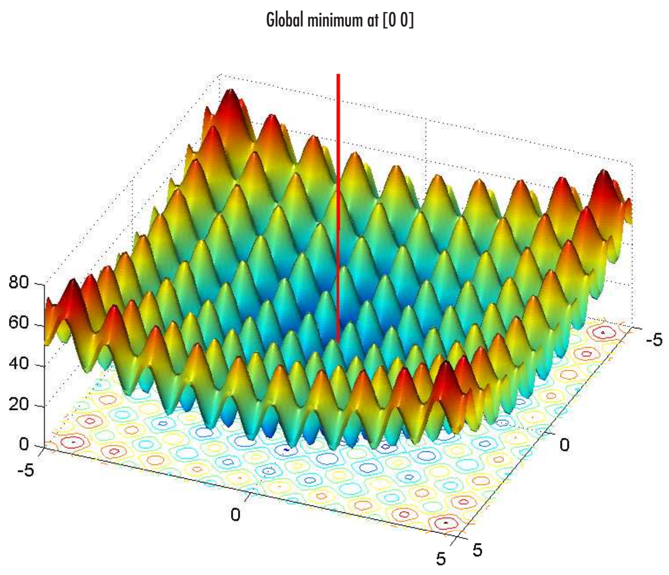
Rastrigin's Function

This section presents an example that shows how to find the minimum of Rastrigin's function, a function that is often used to test the genetic algorithm.

For two independent variables, Rastrigin's function is defined as

$$Ras(x) = 20 + x_1^2 + x_2^2 - 10\left(\cos 2\pi x_1 + \cos 2\pi x_2\right).$$

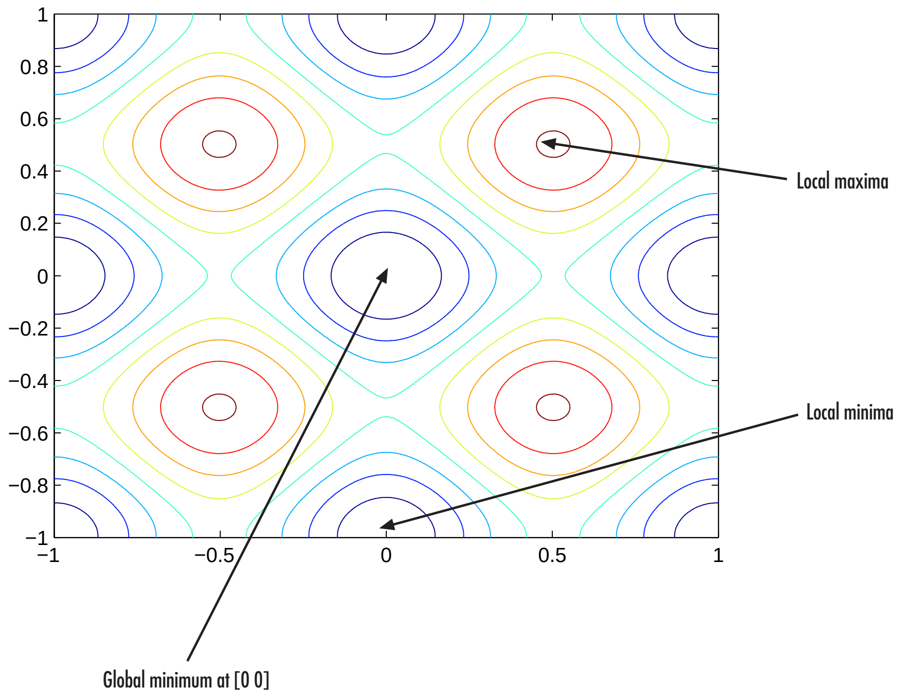
Global Optimization Toolbox software contains the `rastriginsfcn.m` file, which computes the values of Rastrigin's function. The following figure shows a plot of Rastrigin's function.



As the plot shows, Rastrigin's function has many local minima—the “valleys” in the plot. However, the function has just one global minimum, which occurs at the point [0 0] in the *x-y* plane, as indicated by the vertical line in the plot, where the value of the function is 0. At any local minimum other than [0 0], the value of Rastrigin's function is greater than 0. The farther the local minimum is from the origin, the larger the value of the function is at that point.

Rastrigin's function is often used to test the genetic algorithm, because its many local minima make it difficult for standard, gradient-based methods to find the global minimum.

The following contour plot of Rastrigin's function shows the alternating maxima and minima.



Finding the Minimum of Rastrigin's Function

This section explains how to find the minimum of Rastrigin's function using the genetic algorithm.

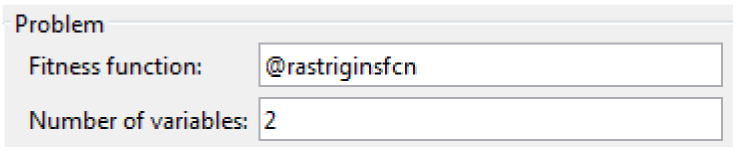
Note Because the genetic algorithm uses random number generators, the algorithm returns slightly different results each time you run it.

To find the minimum, do the following steps:

1. Enter `optimtool('ga')` at the command line to open the Optimization app.
2. Enter the following in the Optimization app:
   - In the **Fitness function** field, enter `@rastriginsfcn`.

- In the **Number of variables** field, enter 2, the number of independent variables for Rastrigin's function.

  The **Fitness function** and **Number of variables** fields should appear as shown in the following figure.

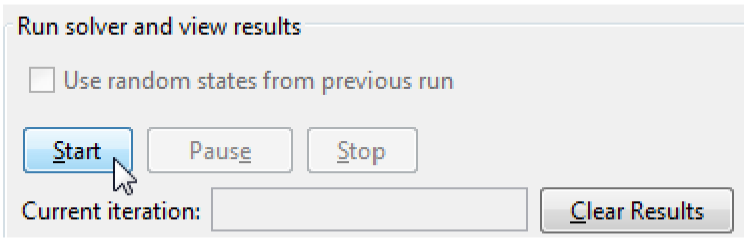


Problem

Fitness function: @rastriginsfcn

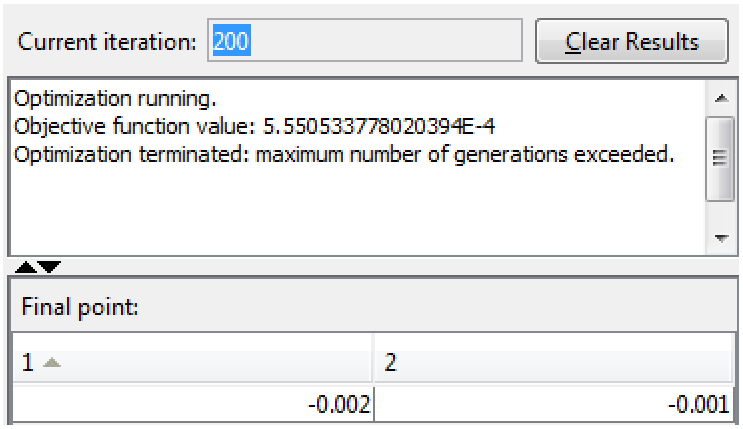
Number of variables: 2

3 Click the **Start** button in the **Run solver and view results** pane, as shown in the following figure.


Run solver and view results
Use random states from previous run
Start
Pause
Stop
Current iteration:
Clear Results

While the algorithm is running, the **Current iteration** field displays the number of the current generation. You can temporarily pause the algorithm by clicking the **Pause** button. When you do so, the button name changes to **Resume**. To resume the algorithm from the point at which you paused it, click **Resume**.

When the algorithm is finished, the **Run solver and view results** pane appears as shown in the following figure. Your numerical results might differ from those in the figure, since `ga` is stochastic.



```
Optimization running.
Objective function value: 5.550533778020394E-4
Optimization terminated: maximum number of generations exceeded.
```

Final point:

1	2
-0.002	-0.001

The display shows:

- The final value of the fitness function when the algorithm terminated:

  ```
  Objective function value: 5.550533778020394E-4
  ```

  Note that the value shown is very close to the actual minimum value of Rastrigin's function, which is `0`. "Setting the Initial Range" on page 5-73, "Setting the Amount of Mutation" on page 5-88, and "Set Maximum Number of Generations" on page 5-108 describe some ways to get a result that is closer to the actual minimum.
- The reason the algorithm terminated.

  ```
  Optimization terminated: maximum number of generations exceeded.
  ```
- The final point, which in this example is `[-0.002 -0.001]`.

Finding the Minimum from the Command Line

To find the minimum of Rastrigin's function from the command line, enter

```
rng(1,'twister') % for reproducibility
[x fval exitflag] = ga(@rastriginsfcn, 2)
```

This returns

```
Optimization terminated:
```

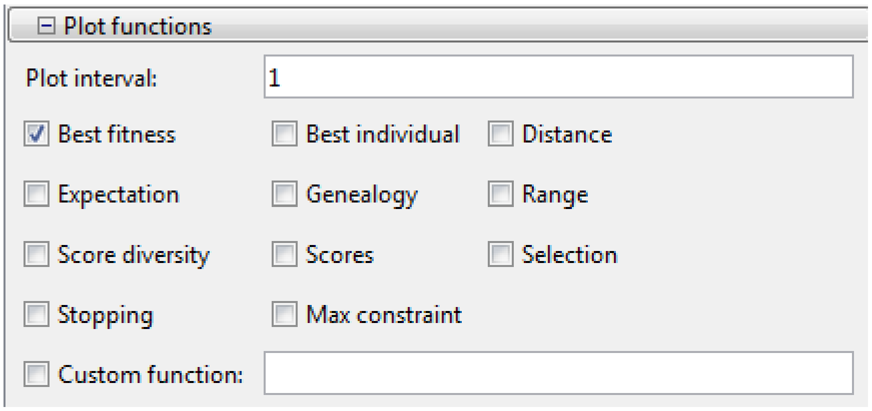
```
average change in the fitness value less than options.TolFun.

x =
   -0.0017   -0.0185

fval =
    0.0682

exitflag =
     1
```

- `x` is the final point returned by the algorithm.
- `fval` is the fitness function value at the final point.
- `exitflag` is integer value corresponding to the reason that the algorithm terminated.

Note: Because the genetic algorithm uses random number generators, the algorithm returns slightly different results each time you run it.

Displaying Plots

The Optimization app **Plot functions** pane enables you to display various plots that provide information about the genetic algorithm while it is running. This information can help you change options to improve the performance of the algorithm. For example, to plot the best and mean values of the fitness function at each generation, select the box next to **Best fitness**, as shown in the following figure.

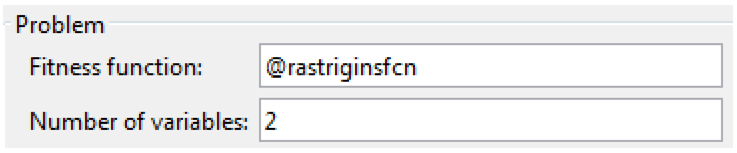

Plot functions

Plot interval: 1

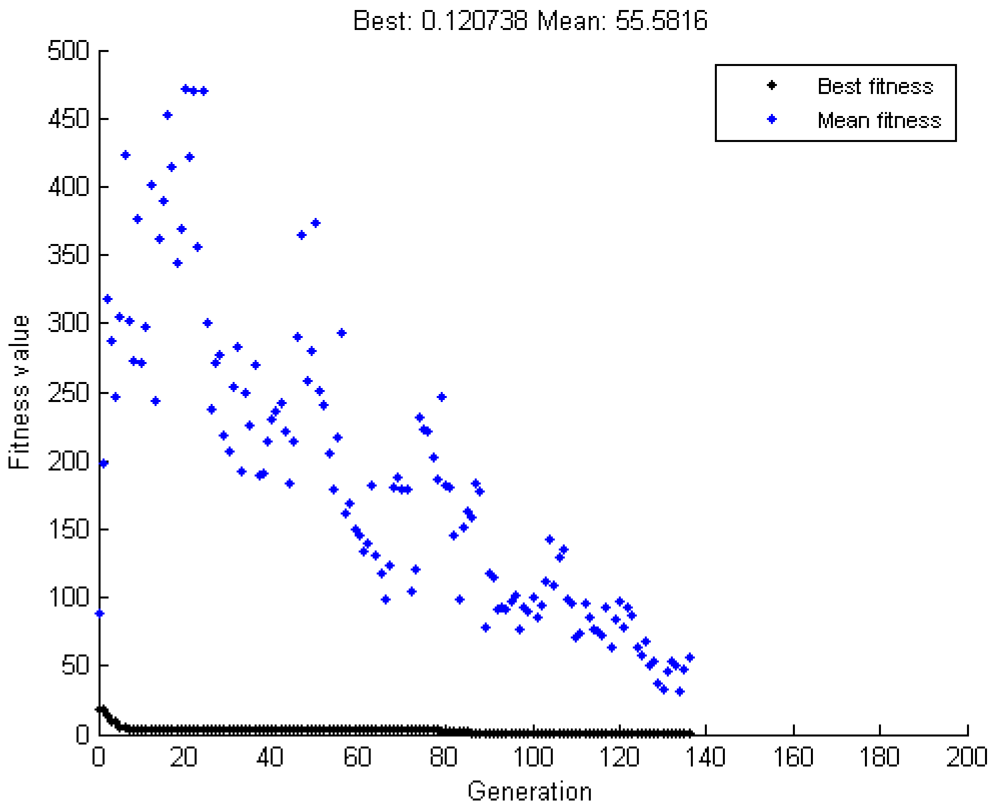
[✓] Best fitness · [ ] Best individual · [ ] Distance
[ ] Expectation · [ ] Genealogy · [ ] Range
[ ] Score diversity · [ ] Scores · [ ] Selection
[ ] Stopping · [ ] Max constraint
[ ] Custom function:

When you click **Start**, the Optimization app displays a plot of the best and mean values of the fitness function at each generation.

Try this on “Minimize Rastrigin's Function” on page 5-6:


Problem
Fitness function: @rastriginsfcn
Number of variables: 2

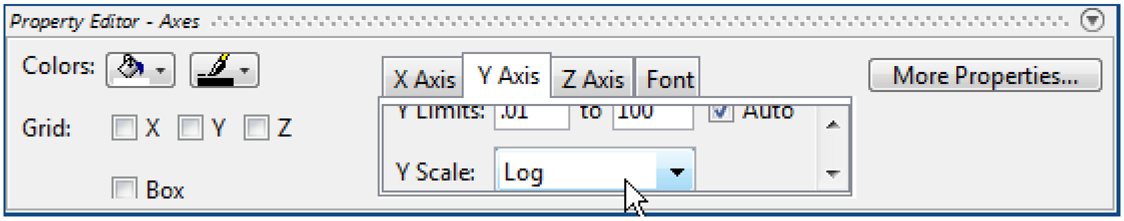
When the algorithm stops, the plot appears as shown in the following figure.



The points at the bottom of the plot denote the best fitness values, while the points above them denote the averages of the fitness values in each generation. The plot also displays the best and mean values in the current generation numerically at the top.

To get a better picture of how much the best fitness values are decreasing, you can change the scaling of the *y*-axis in the plot to logarithmic scaling. To do so,

1. Select **Axes Properties** from the **Edit** menu in the plot window to open the Property Editor attached to your figure window as shown below.

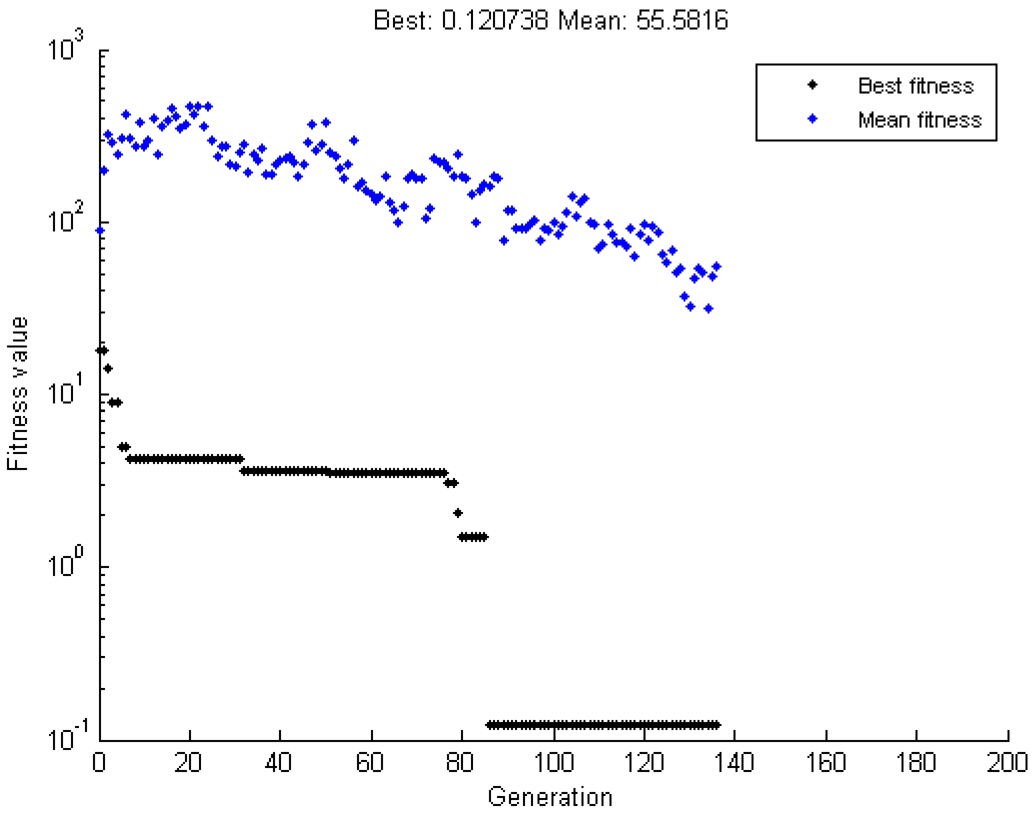

Property Editor - Axes

Y Limits: .01 to 100 ✓ Auto

Y Scale: Log

2. Click the **Y Axis** tab.
3. In the **Y Scale** pane, select **Log**.

   The plot now appears as shown in the following figure.



Typically, the best fitness value improves rapidly in the early generations, when the individuals are farther from the optimum. The best fitness value improves more slowly in later generations, whose populations are closer to the optimal point.

Note When you display more than one plot, you can open a larger version of a plot in a separate window. Right-click (**Ctrl**-click for Mac) on a blank area in a plot while `ga` is running, or after it has stopped, and choose the sole menu item.

"Plot Options" on page 10-29 describes the types of plots you can create.

Genetic Algorithm Terminology

In this section...

Fitness Functions

The *fitness function* is the function you want to optimize. For standard optimization algorithms, this is known as the objective function. The toolbox software tries to find the minimum of the fitness function.

Write the fitness function as a file or anonymous function, and pass it as a function handle input argument to the main genetic algorithm function.

Individuals

An *individual* is any point to which you can apply the fitness function. The value of the fitness function for an individual is its score. For example, if the fitness function is

$$f(x_1, x_2, x_3) = (2x_1 + 1)^2 + (3x_2 + 4)^2 + (x_3 - 2)^2 ,$$

the vector (2, -3, 1), whose length is the number of variables in the problem, is an individual. The score of the individual (2, –3, 1) is *f*(2, –3, 1) = 51.

An individual is sometimes referred to as a *genome* and the vector entries of an individual as *genes*.

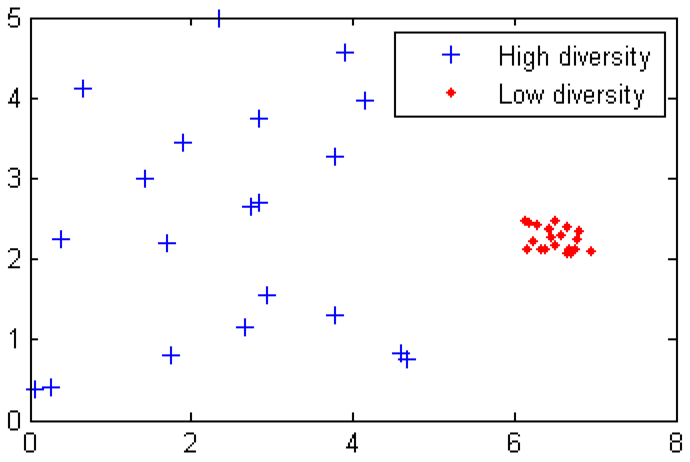
Populations and Generations

A *population* is an array of individuals. For example, if the size of the population is 100 and the number of variables in the fitness function is 3, you represent the population by a 100-by-3 matrix. The same individual can appear more than once in the population. For example, the individual (2, -3, 1) can appear in more than one row of the array.

At each iteration, the genetic algorithm performs a series of computations on the current population to produce a new population. Each successive population is called a new *generation*.

Diversity

Diversity refers to the average distance between individuals in a population. A population has high diversity if the average distance is large; otherwise it has low diversity. In the following figure, the population on the left has high diversity, while the population on the right has low diversity.



Diversity is essential to the genetic algorithm because it enables the algorithm to search a larger region of the space.

Fitness Values and Best Fitness Values

The *fitness value* of an individual is the value of the fitness function for that individual. Because the toolbox software finds the minimum of the fitness function, the *best* fitness value for a population is the smallest fitness value for any individual in the population.

Parents and Children

To create the next generation, the genetic algorithm selects certain individuals in the current population, called *parents*, and uses them to create individuals in the next generation, called *children*. Typically, the algorithm is more likely to select parents that have better fitness values.

How the Genetic Algorithm Works

In this section...

Outline of the Algorithm

The following outline summarizes how the genetic algorithm works:

1 The algorithm begins by creating a random initial population.

2 The algorithm then creates a sequence of new populations. At each step, the algorithm uses the individuals in the current generation to create the next population. To create the new population, the algorithm performs the following steps:

   a Scores each member of the current population by computing its fitness value.

   b Scales the raw fitness scores to convert them into a more usable range of values.

   c Selects members, called parents, based on their fitness.

   d Some of the individuals in the current population that have lower fitness are chosen as *elite*. These elite individuals are passed to the next population.

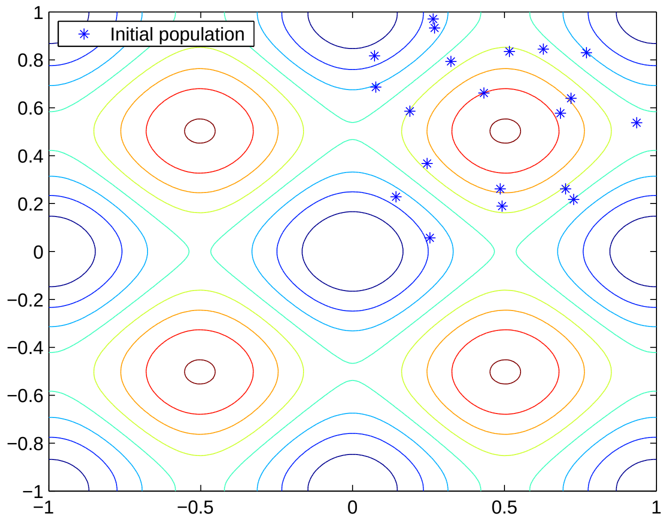
   e Produces children from the parents. Children are produced either by making random changes to a single parent—*mutation*—or by combining the vector entries of a pair of parents—*crossover*.

   f Replaces the current population with the children to form the next generation.

3 The algorithm stops when one of the stopping criteria is met. See "Stopping Conditions for the Algorithm" on page 5-22.

Initial Population

The algorithm begins by creating a random initial population, as shown in the following figure.



In this example, the initial population contains `20` individuals, which is the default value of **Population size** in the **Population** options. Note that all the individuals in the initial population lie in the upper-right quadrant of the picture, that is, their coordinates lie between 0 and 1, because the default value of **Initial range** in the **Population** options is `[0;1]`.

If you know approximately where the minimal point for a function lies, you should set **Initial range** so that the point lies near the middle of that range. For example, if you believe that the minimal point for Rastrigin's function is near the point `[0 0]`, you could set **Initial range** to be `[-1;1]`. However, as this example shows, the genetic algorithm can find the minimum even with a less than optimal choice for **Initial range**.

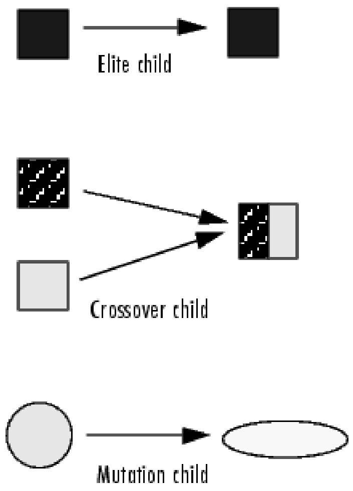
Creating the Next Generation

At each step, the genetic algorithm uses the current population to create the children that make up the next generation. The algorithm selects a group of individuals in the current population, called *parents*, who contribute their *genes*—the entries of their vectors—to their children. The algorithm usually selects individuals that have better fitness values as parents. You can specify the function that the algorithm uses to select the parents in the **Selection function** field in the **Selection** options.

The genetic algorithm creates three types of children for the next generation:

- *Elite children* are the individuals in the current generation with the best fitness values. These individuals automatically survive to the next generation.
- *Crossover children* are created by combining the vectors of a pair of parents.
- *Mutation children* are created by introducing random changes, or mutations, to a single parent.

The following schematic diagram illustrates the three types of children.



"Mutation and Crossover" on page 5-26 explains how to specify the number of children of each type that the algorithm generates and the functions it uses to perform crossover and mutation.

The following sections explain how the algorithm creates crossover and mutation children.

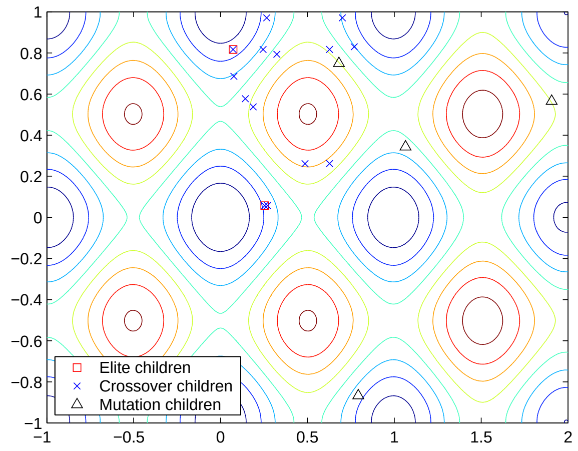
Crossover Children

The algorithm creates crossover children by combining pairs of parents in the current population. At each coordinate of the child vector, the default crossover function randomly selects an entry, or *gene*, at the same coordinate from one of the two parents and assigns it to the child. For problems with linear constraints, the default crossover function creates the child as a random weighted average of the parents.

Mutation Children

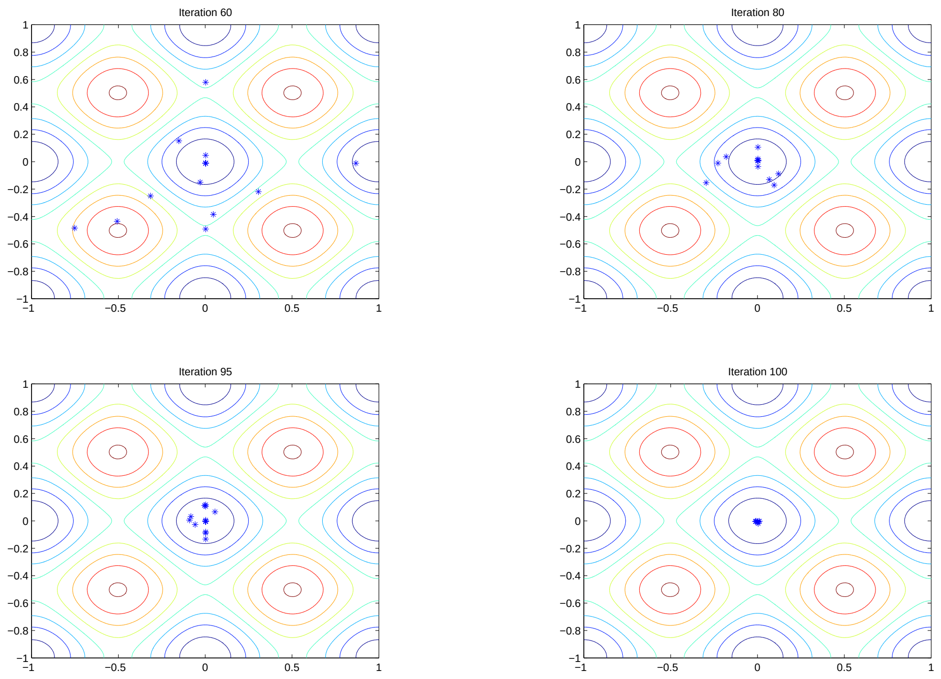
The algorithm creates mutation children by randomly changing the genes of individual parents. By default, for unconstrained problems the algorithm adds a random vector from a Gaussian distribution to the parent. For bounded or linearly constrained problems, the child remains feasible.

The following figure shows the children of the initial population, that is, the population at the second generation, and indicates whether they are elite, crossover, or mutation children.



Plots of Later Generations

The following figure shows the populations at iterations 60, 80, 95, and 100.



As the number of generations increases, the individuals in the population get closer together and approach the minimum point `[0 0]`.

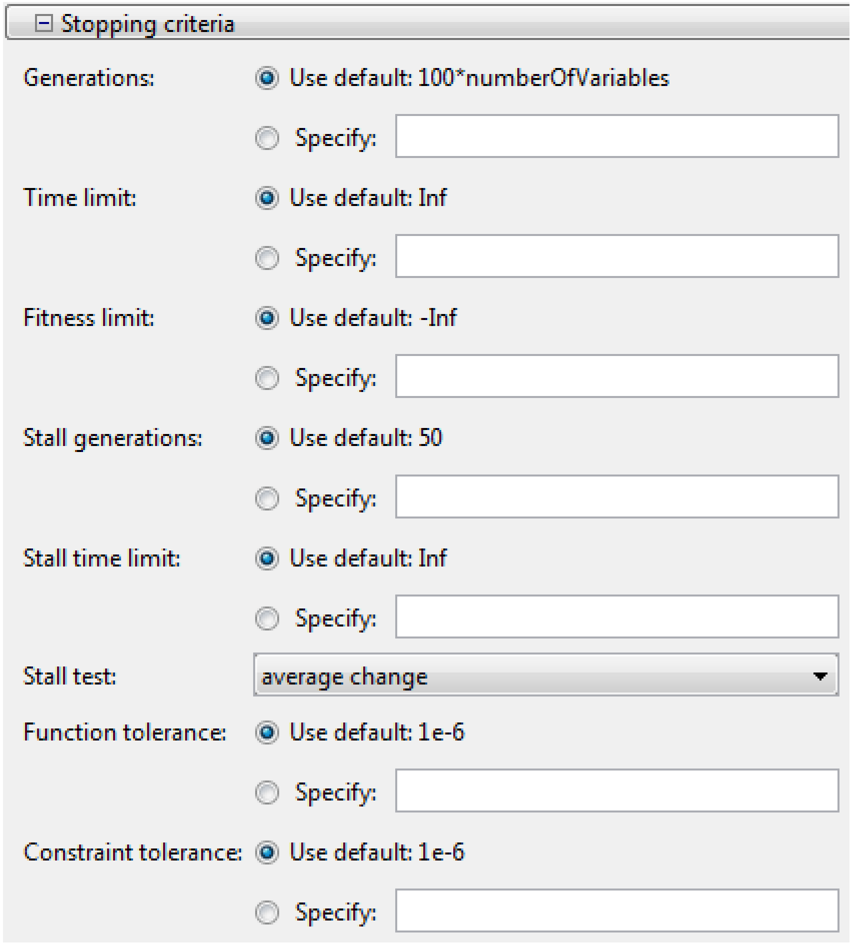
Stopping Conditions for the Algorithm

The genetic algorithm uses the following conditions to determine when to stop:

- **Generations** — The algorithm stops when the number of generations reaches the value of **Generations**.
- **Time limit** — The algorithm stops after running for an amount of time in seconds equal to **Time limit**.
- **Fitness limit** — The algorithm stops when the value of the fitness function for the best point in the current population is less than or equal to **Fitness limit**.
- **Stall generations** — The algorithm stops when the average relative change in the fitness function value over **Stall generations** is less than **Function tolerance**.

- **Stall time limit** — The algorithm stops if there is no improvement in the objective function during an interval of time in seconds equal to **Stall time limit**.
- **Stall test** — The stall condition is either `average change` or `geometric weighted`. For `geometric weighted`, the weighting function is $1/2^n$, where n is the number of generations prior to the current. Both stall conditions apply to the relative change in the fitness function over **Stall generations**.
- **Function Tolerance** — The algorithm runs until the average relative change in the fitness function value over **Stall generations** is less than **Function tolerance**.
- **Nonlinear constraint tolerance** — The **Nonlinear constraint tolerance** is not used as stopping criterion. It is used to determine the feasibility with respect to nonlinear constraints. Also, a point is feasible with respect to linear constraints when the constraint violation is below the square root of **Nonlinear constraint tolerance**.

The algorithm stops as soon as any one of these conditions is met. You can specify the values of these criteria in the **Stopping criteria** pane in the Optimization app. The default values are shown in the pane.



Stopping criteria	
Generations:	[✓ Use default: 100*numberOfVariables] Specify:
Time limit:	[✓ Use default: Inf] Specify:
Fitness limit:	[✓ Use default: -Inf] Specify:
Stall generations:	[✓ Use default: 50] Specify:
Stall time limit:	[✓ Use default: Inf] Specify:
Stall test:	average change
Function tolerance:	[✓ Use default: 1e-6] Specify:
Constraint tolerance:	[✓ Use default: 1e-6] Specify:

When you run the genetic algorithm, the **Run solver and view results** panel displays the criterion that caused the algorithm to stop.

The options **Stall time limit** and **Time limit** prevent the algorithm from running too long. If the algorithm stops due to one of these conditions, you might improve your results by increasing the values of **Stall time limit** and **Time limit**.

Selection

The selection function chooses parents for the next generation based on their scaled values from the fitness scaling function. An individual can be selected more than once as a parent, in which case it contributes its genes to more than one child. The default selection option, `Stochastic uniform`, lays out a line in which each parent corresponds to a section of the line of length proportional to its scaled value. The algorithm moves along the line in steps of equal size. At each step, the algorithm allocates a parent from the section it lands on.

A more deterministic selection option is `Remainder`, which performs two steps:

- In the first step, the function selects parents deterministically according to the integer part of the scaled value for each individual. For example, if an individual's scaled value is 2.3, the function selects that individual twice as a parent.
- In the second step, the selection function selects additional parents using the fractional parts of the scaled values, as in stochastic uniform selection. The function lays out a line in sections, whose lengths are proportional to the fractional part of the scaled value of the individuals, and moves along the line in equal steps to select the parents.

  Note that if the fractional parts of the scaled values all equal 0, as can occur using `Top` scaling, the selection is entirely deterministic.

For details and more selection options, see “Selection Options” on page 10-37.

Reproduction Options

Reproduction options control how the genetic algorithm creates the next generation. The options are

- **Elite count** — The number of individuals with the best fitness values in the current generation that are guaranteed to survive to the next generation. These individuals are called *elite children*. The default value of **Elite count** is `2`.

  When **Elite count** is at least 1, the best fitness value can only decrease from one generation to the next. This is what you want to happen, since the genetic algorithm minimizes the fitness function. Setting **Elite count** to a high value causes the fittest individuals to dominate the population, which can make the search less effective.
- **Crossover fraction** — The fraction of individuals in the next generation, other than elite children, that are created by crossover. “Setting the Crossover Fraction” on page

5-90 describes how the value of **Crossover fraction** affects the performance of the genetic algorithm.

Mutation and Crossover

The genetic algorithm uses the individuals in the current generation to create the children that make up the next generation. Besides elite children, which correspond to the individuals in the current generation with the best fitness values, the algorithm creates

- Crossover children by selecting vector entries, or genes, from a pair of individuals in the current generation and combines them to form a child
- Mutation children by applying random changes to a single individual in the current generation to create a child

Both processes are essential to the genetic algorithm. Crossover enables the algorithm to extract the best genes from different individuals and recombine them into potentially superior children. Mutation adds to the diversity of a population and thereby increases the likelihood that the algorithm will generate individuals with better fitness values.

See "Creating the Next Generation" on page 5-19 for an example of how the genetic algorithm applies mutation and crossover.

You can specify how many of each type of children the algorithm creates as follows:

- **Elite count**, in **Reproduction** options, specifies the number of elite children.
- **Crossover fraction**, in **Reproduction** options, specifies the fraction of the population, other than elite children, that are crossover children.

For example, if the **Population size** is `20`, the **Elite count** is `2`, and the **Crossover fraction** is `0.8`, the numbers of each type of children in the next generation are as follows:

- There are two elite children.
- There are 18 individuals other than elite children, so the algorithm rounds 0.8*18 = 14.4 to 14 to get the number of crossover children.
- The remaining four individuals, other than elite children, are mutation children.

Mixed Integer Optimization

In this section...

Solving Mixed Integer Optimization Problems

`ga` can solve problems when certain variables are integer-valued. Give `IntCon`, a vector of the *x* components that are integers:

```
[x,fval,exitflag] = ga(fitnessfcn,nvars,A,b,[],[],...
    lb,ub,nonlcon,IntCon,options)
```

`IntCon` is a vector of positive integers that contains the *x* components that are integer-valued. For example, if you want to restrict `x(2)` and `x(10)` to be integers, set `IntCon` to `[2,10]`.

Note: Restrictions exist on the types of problems that `ga` can solve with integer variables. In particular, `ga` does not accept any equality constraints when there are integer variables. For details, see "Characteristics of the Integer ga Solver" on page 5-29.

Tip `ga` solves integer problems best when you provide lower and upper bounds for every *x* component.

Mixed Integer Optimization of Rastrigin's Function

This example shows how to find the minimum of Rastrigin's function restricted so the first component of *x* is an integer. The components of *x* are further restricted to be in the region $5\pi \leq x(1) \leq 20\pi,\ \ -20\pi \leq x(2) \leq -4\pi$.

Set up the bounds for your problem

```
lb = [5*pi,-20*pi];
ub = [20*pi,-4*pi];
```

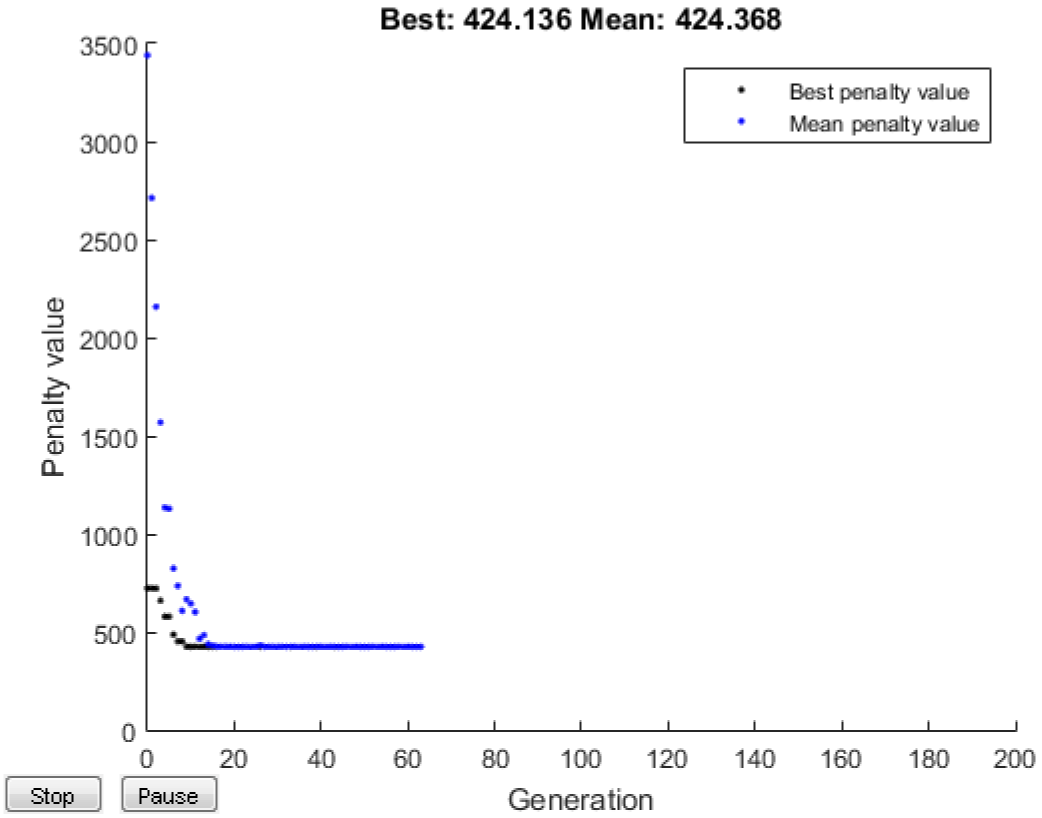
Set a plot function so you can view the progress of ga

```
opts = gaoptimset('PlotFcns',@gaplotbestf);
```

Call the ga solver where x(1) has integer values

```
rng(1,'twister') % for reproducibility
IntCon = 1;
[x,fval,exitflag] = ga(@rastriginsfcn,2,[],[],[],[],...
    lb,ub,[],IntCon,opts)
```

```
Optimization terminated: average change in the penalty fitness value less than options
and constraint violation is less than options.TolCon.

x =

   16.0000  -12.9325


fval =

  424.1355


exitflag =

     1
```



ga converges quickly to the solution.

Characteristics of the Integer ga Solver

There are some restrictions on the types of problems that `ga` can solve when you include integer constraints:

- No linear equality constraints. You must have `Aeq = []` and `beq = []`. For a possible workaround, see "No Equality Constraints" on page 5-30.
- No nonlinear equality constraints. Any nonlinear constraint function must return `[]` for the nonlinear equality constraint. For a possible workaround, see "Example: Integer Programming with a Nonlinear Equality Constraint" on page 5-30.

- Only `doubleVector` population type.
- No custom creation function (`CreationFcn` option), crossover function (`CrossoverFcn` option), mutation function (`MutationFcn` option), or initial scores (`InitialScores` option). If you supply any of these, `ga` overrides their settings.
- `ga` uses only the binary tournament selection function (`SelectionFcn` option), and overrides any other setting.
- No hybrid function. `ga` overrides any setting of the `HybridFcn` option.
- `ga` ignores the `ParetoFraction`, `DistanceMeasureFcn`, `InitialPenalty`, and `PenaltyFactor` options.

The listed restrictions are mainly natural, not arbitrary. For example:

- There are no hybrid functions that support integer constraints. So `ga` does not use hybrid functions when there are integer constraints.
- To obtain integer variables, `ga` uses special creation, crossover, and mutation functions.

No Equality Constraints

You cannot use equality constraints and integer constraints in the same problem. You can try to work around this restriction by including two inequality constraints for each linear equality constraint. For example, to try to include the constraint
$3x_1 – 2x_2 = 5$,

create two inequality constraints:
$3x_1 – 2x_2 \le 5$
$3x_1 – 2x_2 \ge 5$.

To write these constraints in the form `A x` ≤ `b`, multiply the second inequality by `-1`:
$–3x_1 + 2x_2 \le –5$.

You can try to include the equality constraint using `A` = `[3,-2;-3,2]` and `b` = `[5;-5]`.

Be aware that this procedure can fail; `ga` has difficulty with simultaneous integer and equality constraints.

Example: Integer Programming with a Nonlinear Equality Constraint

This example attempts to locate the minimum of the Ackley function in five dimensions with these constraints:

- `x(1)`, `x(3)`, and `x(5)` are integers.

- `norm(x) = 4`.

The Ackley function, described briefly in "Resuming ga From the Final Population" on page 5-57, is difficult to minimize. Adding integer and equality constraints increases the difficulty.

To include the nonlinear equality constraint, give a small tolerance `tol` that allows the norm of `x` to be within `tol` of `4`. Without a tolerance, the nonlinear equality constraint is never satisfied, and the solver does not realize when it has a feasible solution.

1. Write the expression `norm(x) = 4` as two "less than zero" inequalities:
   `norm(x) - 4` $\leq$ `0`
   `-(norm(x) - 4)` $\leq$ `0`.
2. Allow a small tolerance in the inequalities:
   `norm(x) - 4 - tol` $\leq$ `0`
   `-(norm(x) - 4) - tol` $\leq$ `0`.
3. Write a nonlinear inequality constraint function that implements these inequalities:

   ```
   function [c, ceq] = eqCon(x)

   ceq = [];
   rad = 4;
   tol = 1e-3;
   confcnval = norm(x) - rad;
   c = [confcnval - tol;-confcnval - tol];
   ```
4. Set options:

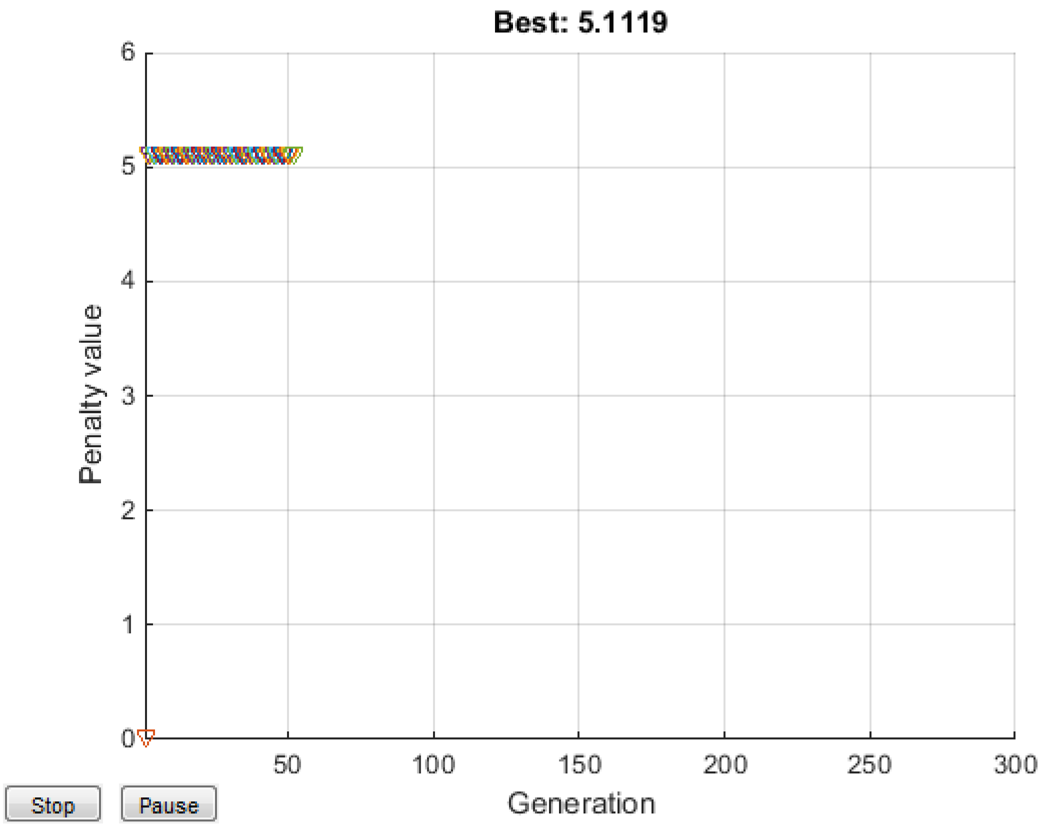
   - `StallGenLimit = 50` — Allow the solver to try for a while.
   - `TolFun = 1e-10` — Specify a stricter stopping criterion than usual.
   - `Generations = 300` — Allow more generations than default.
   - `PlotFcns = @gaplotbestfun` — Observe the optimization.

   ```
   opts = gaoptimset('StallGenLimit',50,'TolFun',1e-10,...
       'Generations',300,'PlotFcns',@gaplotbestfun);
   ```
5. Set lower and upper bounds to help the solver:

   ```
   nVar = 5;
   lb = -5*ones(1,nVar);
   ub = 5*ones(1,nVar);
   ```
6. Solve the problem:

   ```
   rng(1,'twister') % for reproducibility
   ```

```
[x,fval,exitflag] = ga(@ackleyfcn,nVar,[],[],[],[], ...
    lb,ub,@eqCon,[1 3 5],opts);

Optimization terminated: stall generations limit exceeded
and constraint violation is less than options.TolCon.
```



7 Examine the solution:

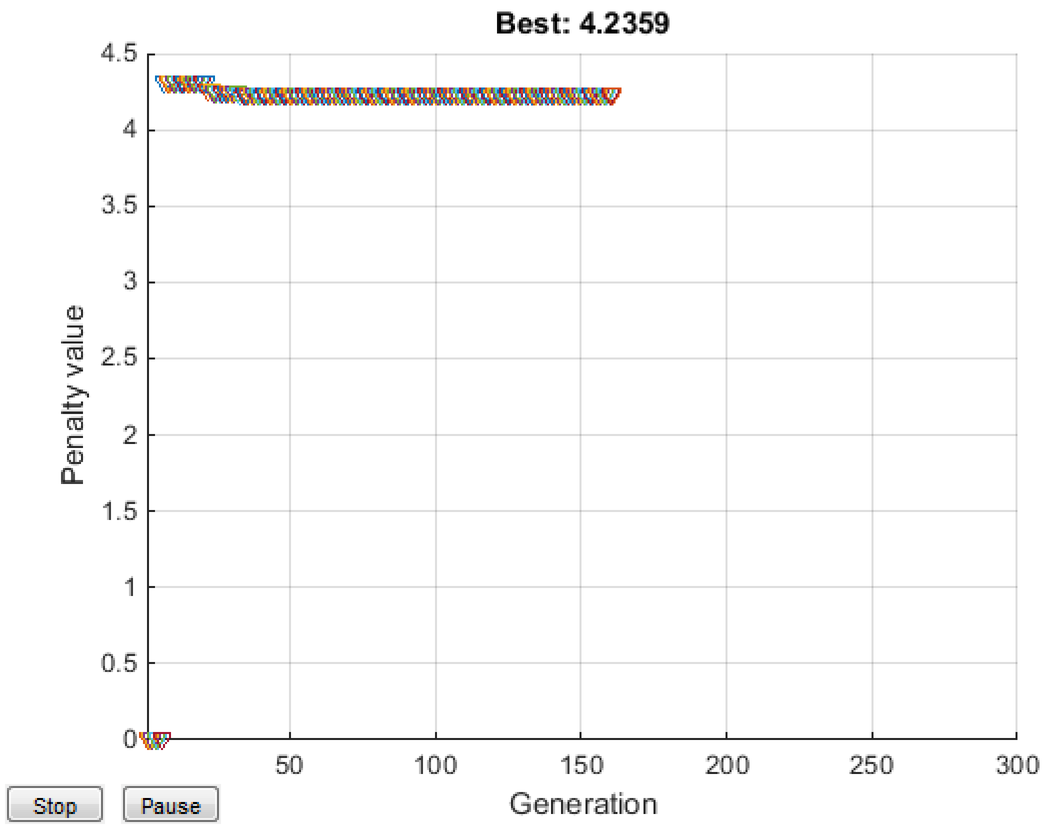
```
x,fval,exitflag,norm(x)

x =

   -3.0000    0.8233    1.0000   -1.1503    2.0000
```

```
fval =

    5.1119


exitflag =

     3


ans =

    4.0001
```

The odd `x` components are integers, as specified. The norm of `x` is `4`, to within the given relative tolerance of `1e-3`.

8 Despite the positive exit flag, the solution is not the global optimum. Run the problem again and examine the solution:

```
opts = gaoptimset(opts,'Display','off');
[x2,fval2,exitflag2] = ga(@ackleyfcn,nVar,[],[],[],[], ...
    lb,ub,@eqCon,[1 3 5],opts);
```



Examine the second solution:

```
x2,fval2,exitflag2,norm(x2)

x2 =

   -1.0000   -0.9959   -2.0000    0.9960    3.0000


fval2 =

    4.2359
```

```
exitflag2 =

     1


ans =

    3.9980
```

The second run gives a better solution (lower fitness function value). Again, the odd `x` components are integers, and the norm of `x2` is `4`, to within the given relative tolerance of `1e-3`.

Be aware that this procedure can fail; `ga` has difficulty with simultaneous integer and equality constraints.

Integer ga Algorithm

Integer programming with `ga` involves several modifications of the basic algorithm (see "How the Genetic Algorithm Works" on page 5-18). For integer programming:

- Special creation, crossover, and mutation functions enforce variables to be integers. For details, see Deep et al. [2].
- The genetic algorithm attempts to minimize a penalty function, not the fitness function. The penalty function includes a term for infeasibility. This penalty function is combined with binary tournament selection to select individuals for subsequent generations. The penalty function value of a member of a population is:
  - If the member is feasible, the penalty function is the fitness function.
  - If the member is infeasible, the penalty function is the maximum fitness function among feasible members of the population, plus a sum of the constraint violations of the (infeasible) point.

  For details of the penalty function, see Deb [1].

References

[1] Deb, Kalyanmoy. *An efficient constraint handling method for genetic algorithms.* Computer Methods in Applied Mechanics and Engineering, 186(2–4), pp. 311–338, 2000.

[2] Deep, Kusum, Krishna Pratap Singh, M.L. Kansal, and C. Mohan. *A real coded genetic algorithm for solving integer and mixed integer optimization problems.* Applied Mathematics and Computation, 212(2), pp. 505–518, 2009.

Solving a Mixed Integer Engineering Design Problem Using the Genetic Algorithm

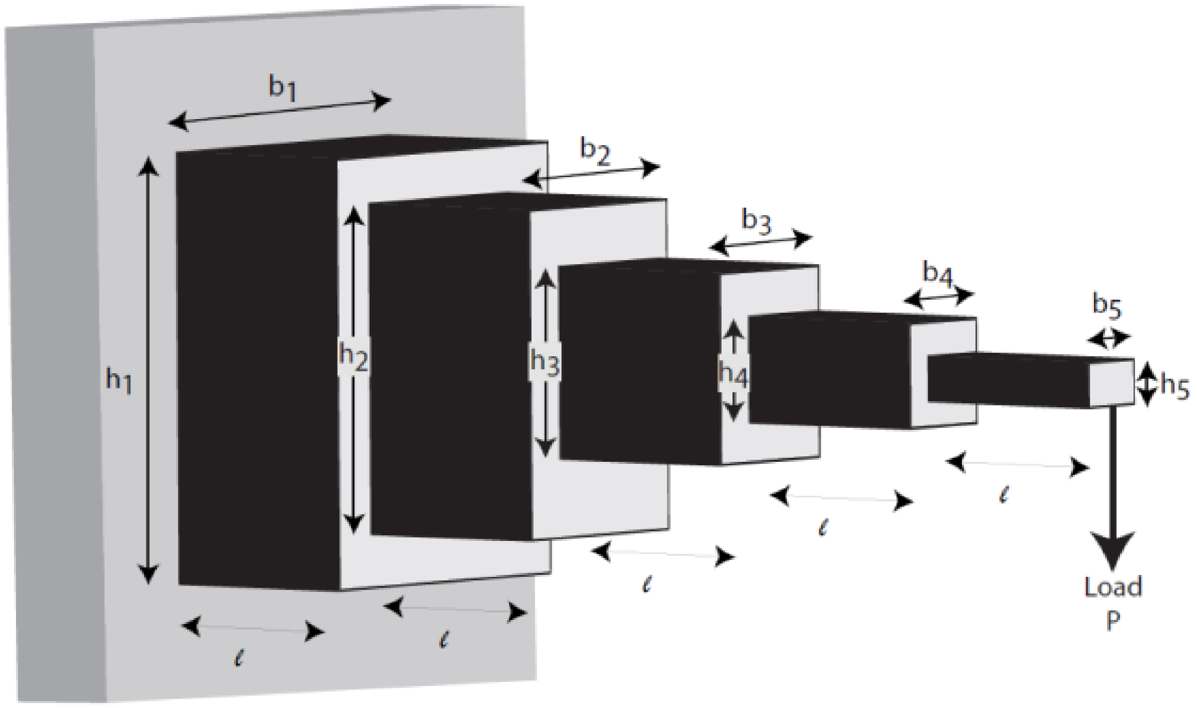
This example shows how to solve a mixed integer engineering design problem using the Genetic Algorithm (`ga`) solver in Global Optimization Toolbox.

The problem illustrated in this example involves the design of a stepped cantilever beam. In particular, the beam must be able to carry a prescribed end load. We will solve a problem to minimize the beam volume subject to various engineering design constraints.

In this example we will solve two bounded versions of the problem published in [1].

Stepped Cantilever Beam Design Problem

A stepped cantilever beam is supported at one end and a load is applied at the free end, as shown in the figure below. The beam must be able to support the given load, P, at a fixed distance L from the support. Designers of the beam can vary the width (b_i) and height (h_i) of each section. We will assume that each section of the cantilever has the same length, l.



Volume of the beam

The volume of the beam, V, is the sum of the volume of the individual sections

$$V = l(b_1h_1 + b_2h_2 + b_3h_3 + b_4h_4 + b_5h_5)$$

Constraints on the Design : 1 - Bending Stress

Consider a single cantilever beam, with the centre of coordinates at the centre of its cross section at the free end of the beam. The bending stress at a point (x, y, z) in the beam is given by the following equation

$$\sigma_b = M(x)y/I$$

where $M(x)$ is the bending moment at x, x is the distance from the end load and I is the area moment of inertia of the beam.

Now, in the stepped cantilever beam shown in the figure, the maximum moment of each section of the beam is PD_i, where D_i is the maximum distance from the end load, P, for each section of the beam. Therefore, the maximum stress for the i-th section of the beam, σ_i, is given by

$$\sigma_i = PD_i(h_i/2)/I_i$$

where the maximum stress occurs at the edge of the beam, $y = h_i/2$. The area moment of inertia of the i-th section of the beam is given by

$$I_i = b_i h_i^3/12$$

Substituting this into the equation for σ_i gives

$$\sigma_i = 6PD_i/b_i h_i^2$$

The bending stress in each part of the cantilever should not exceed the maximum allowable stress, σ_{max}. Consequently, we can finally state the five bending stress constraints (one for each step of the cantilever)

$$\frac{6Pl}{b_5 h_5^2} \leq \sigma_{max}$$

$$\frac{6P(2l)}{b_4 h_4^2} \leq \sigma_{max}$$

$$\frac{6P(3l)}{b_3 h_3^2} \leq \sigma_{max}$$

$$\frac{6P(4l)}{b_2 h_2^2} \leq \sigma_{max}$$

$$\frac{6P(5l)}{b_1 h_1^2} \leq \sigma_{max}$$

Constraints on the Design : 2 - End deflection

The end deflection of the cantilever can be calculated using Castigliano's second theorem, which states that

$$\delta = \frac{\partial U}{\partial P}$$

where δ is the deflection of the beam, U is the energy stored in the beam due to the applied force, P.

The energy stored in a cantilever beam is given by

$$U = \int_0^L M^2/2EI \, \mathrm{d}x$$

where M is the moment of the applied force at x.

Given that $M = Px$ for a cantilever beam, we can write the above equation as

$$U = P^2/2E \int_0^l [(x+4l)^2/I_1 + (x+3l)^2/I_2 + (x+2l)^2/I_3 + (x+l)^2/I_4 + x^2/I_5] \, \mathrm{d}x$$

where I_n is the area moment of inertia of the n-th part of the cantilever. Evaluating the integral gives the following expression for U.

$$U = (P^2/2)(l^3/3E)(61/I_1 + 37/I_2 + 19/I_3 + 7/I_4 + 1/I_5)$$

Applying Castigliano's theorem, the end deflection of the beam is given by

$$\delta = Pl^3/3E(61/I_1 + 37/I_2 + 19/I_3 + 7/I_4 + 1/I_5)$$

Now, the end deflection of the cantilever, δ, should be less than the maximum allowable deflection, δ_{max}, which gives us the following constraint.

$$Pl^3/3E(61/I_1 + 37/I_2 + 19/I_3 + 7/I_4 + 1/I_5) \leq \delta_{max}$$

Constraints on the Design : 3 - Aspect ratio

For each step of the cantilever, the aspect ratio must not exceed a maximum allowable aspect ratio, a_{max}. That is,

$h_i/b_i \leq a_{max}$ for $i = 1, ..., 5$

State the Optimization Problem

We are now able to state the problem to find the optimal parameters for the stepped cantilever beam given the stated constraints.

Let $x_1 = b_1$, $x_2 = h_1$, $x_3 = b_2$, $x_4 = h_2$, $x_5 = b_3$, $x_6 = h_3$, $x_7 = b_4$, $x_8 = h_4$, $x_9 = b_5$ and $x_{10} = h_5$

Minimize:

$$V = l(x_1x_2 + x_3x_4 + x_5x_6 + x_7x_8 + x_9x_{10})$$

Subject to:

$$\frac{6Pl}{x_9x_{10}^2} \leq \sigma_{max}$$

$$\frac{6P(2l)}{x_7x_8^2} \leq \sigma_{max}$$

$$\frac{6P(3l)}{x_5x_6^2} \leq \sigma_{max}$$

$$\frac{6P(4l)}{x_3x_4^2} \leq \sigma_{max}$$

$$\frac{6P(5l)}{x_1x_2^2} \leq \sigma_{max}$$

$$\frac{Pl^3}{E}\left(\frac{244}{x_1x_2^3}+\frac{148}{x_3x_4^3}+\frac{76}{x_5x_6^3}+\frac{28}{x_7x_8^3}+\frac{4}{x_9x_{10}^3}\right)\le\delta_{max}$$

$$x_2/x_1 \le 20,\ x_4/x_3 \le 20,\ x_6/x_5 \le 20,\ x_8/x_7 \le 20 \text{ and } x_{10}/x_9 \le 20$$

The first step of the beam can only be machined to the nearest centimetre. That is, x_1 and x_2 must be integer. The remaining variables are continuous. The bounds on the variables are given below:-

$$1 \le x_1 \le 5$$

$$30 \le x_2 \le 65$$

$$2.4 \le x_3, x_5 \le 3.1$$

$$45 \le x_4, x_6 \le 60$$

$$1 \le x_7, x_9 \le 5$$

$$30 \le x_8, x_{10} \le 65$$

Design Parameters for this Problem

For the problem we will solve in this example, the end load that the beam must support is $P = 50000N$.

The beam lengths and maximum end deflection are:

- Total beam length, $L = 500cm$
- Individual section of beam, $l = 100cm$
- Maximum beam end deflection, $\delta_{max} = 2.7cm$

The maximum allowed stress in each step of the beam, $\sigma_{max} = 14000N/cm^2$

Young's modulus of each step of the beam, $E = 2 \times 10^7 N/cm^2$

Solve the Mixed Integer Optimization Problem

We now solve the problem described in *State the Optimization Problem*.

Define the Fitness and Constraint Functions

Examine the MATLAB files `cantileverVolume.m` and `cantileverConstraints.m` to see how the fitness and constraint functions are implemented.

A note on the linear constraints: When linear constraints are specified to `ga`, you normally specify them via the `A`, `b`, `Aeq` and `beq` inputs. In this case we have specified them via the nonlinear constraint function. This is because later in this example, some of the variables will become discrete. When there are discrete variables in the problem it is far easier to specify linear constraints in the nonlinear constraint function. The alternative is to modify the linear constraint matrices to work in the transformed variable space, which is not trivial and maybe not possible. Also, in the mixed integer `ga` solver, the linear constraints are not treated any differently to the nonlinear constraints regardless of how they are specified.

Set the Bounds

Create vectors containing the lower bound (`lb`) and upper bound constraints (`ub`).

```
lb = [1 30 2.4 45 2.4 45 1 30 1 30];
ub = [5 65 3.1 60 3.1 60 5 65 5 65];
```

Set the Options

To obtain a more accurate solution, we increase the PopulationSize, and Generations options from their default values, and decrease the EliteCount and TolFun options. These settings cause `ga` to use a larger population (increased PopulationSize), to increase the search of the design space (reduced EliteCount), and to keep going until its best member changes by very little (small TolFun). We also specify a plot function to monitor the penalty function value as `ga` progresses.

Note that there are a restricted set of `ga` options available when solving mixed integer problems - see Global Optimization Toolbox User's Guide for more details.

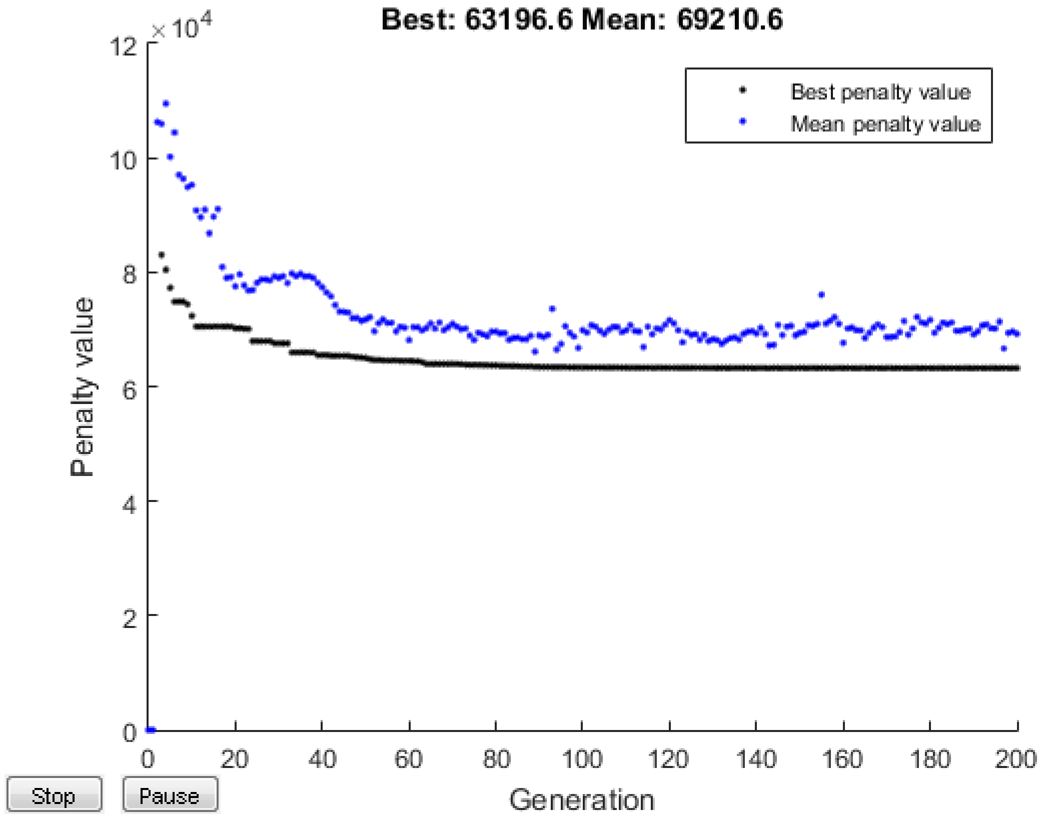
```
opts = gaoptimset(...
    'PopulationSize', 150, ...
    'Generations', 200, ...
    'EliteCount', 10, ...
    'TolFun', 1e-8, ...
    'PlotFcns', @gaplotbestf);
```

Call ga to Solve the Problem

We can now call ga to solve the problem. In the problem statement x_1 and x_2 are integer variables. We specify this by passing the index vector `[1 2]` to ga after the nonlinear constraint input and before the options input. We also seed and set the random number generator here for reproducibility.

```
rng(0, 'twister');
[xbest, fbest, exitflag] = ga(@cantileverVolume, 10, [], [], [], [], ...
    lb, ub, @cantileverConstraints, [1 2], opts);
```

```
Optimization terminated: maximum number of generations exceeded.
```



Analyze the Results

If a problem has integer constraints, `ga` reformulates it internally. In particular, the fitness function in the problem is replaced by a penalty function which handles the constraints. For feasible population members, the penalty function is the same as the fitness function.

The solution returned from `ga` is displayed below. Note that the section nearest the support is constrained to have a width (x_1) and height (x_2) which is an integer value and this constraint has been honored by GA.

```
display(xbest);
```

```
xbest =

  Columns 1 through 7

    3.0000   60.0000    2.8326   56.6516    2.5725   51.4445    2.2126

  Columns 8 through 10

   44.2423    1.7512   34.9805
```

We can also ask `ga` to return the optimal volume of the beam.

```
fprintf('\nCost function returned by ga = %g\n', fbest);
```

```
Cost function returned by ga = 63196.6
```

Add Discrete Non-Integer Variable Constraints

The engineers are now informed that the second and third steps of the cantilever can only have widths and heights that are chosen from a standard set. In this section, we show how to add this constraint to the optimization problem. Note that with the addition of this constraint, this problem is identical to that solved in [1].

First, we state the extra constraints that will be added to the above optimization

- The width of the second and third steps of the beam must be chosen from the following set:- [2.4, 2.6, 2.8, 3.1] cm

- The height of the second and third steps of the beam must be chosen from the following set:- [45, 50, 55, 60] cm

To solve this problem, we need to be able to specify the variables x_3, x_4, x_5 and x_6 as discrete variables. To specify a component x_j as taking discrete values from the set $S = v_1, \ldots, v_k$, optimize with x_j an integer variable taking values from 1 to k, and use $S(x_j)$ as the discrete value. To specify the range (1 to k), set 1 as the lower bound and k as the upper bound.

So, first we transform the bounds on the discrete variables. Each set has 4 members and we will map the discrete variables to an integer in the range [1, 4]. So, to map these variables to be integer, we set the lower bound to 1 and the upper bound to 4 for each of the variables.

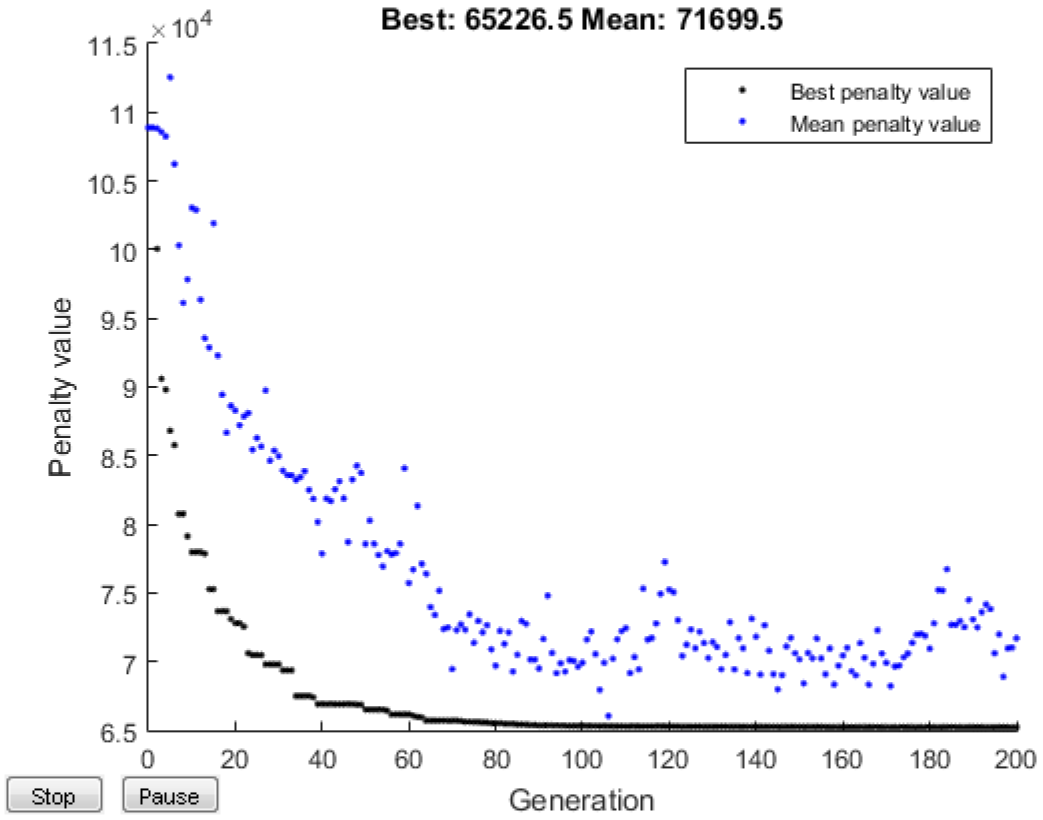
```
lb = [1 30 1 1 1 1 1 30 1 30];
ub = [5 65 4 4 4 4 5 65 5 65];
```

Transformed (integer) versions of x_3, x_4, x_5 and x_6 will now be passed to the fitness and constraint functions when the `ga` solver is called. To evaluate these functions correctly, x_3, x_4, x_5 and x_6 need to be transformed to a member of the given discrete set in these functions. To see how this is done, examine the MATLAB files `cantileverVolumeWithDisc.m`, `cantileverConstraintsWithDisc.m` and `cantileverMapVariables.m`.

Now we can call `ga` to solve the problem with discrete variables. In this case $x_1, \ldots, x_6$ are integers. This means that we pass the index vector `1:6` to `ga` to define the integer variables.

```
rng(0, 'twister');
[xbestDisc, fbestDisc, exitflagDisc] = ga(@cantileverVolumeWithDisc, ...
    10, [], [], [], [], lb, ub, @cantileverConstraintsWithDisc, 1:6, opts);
```

```
Optimization terminated: maximum number of generations exceeded.
```



Analyze the Results

`xbestDisc(3:6)` are returned from `ga` as integers (i.e. in their transformed state). We need to reverse the transform to retrieve the value in their engineering units.

```
xbestDisc = cantileverMapVariables(xbestDisc);
display(xbestDisc);
```

```
xbestDisc =

  Columns 1 through 7

    3.0000    60.0000     3.1000    55.0000     2.8000    50.0000     2.3036
  Columns 8 through 10

   43.6153    1.7509   35.0071
```

As before, the solution returned from `ga` honors the constraint that x_1 and x_2 are integers. We can also see that x_3, x_5 are chosen from the set [2.4, 2.6, 2.8, 3.1] cm and x_4, x_6 are chosen from the set [45, 50, 55, 60] cm.

Recall that we have added additional constraints on the variables `x(3)`, `x(4)`, `x(5)` and `x(6)`. As expected, when there are additional discrete constraints on these variables, the optimal solution has a higher minimum volume. Note further that the solution reported in [1] has a minimum volume of $64558cm^3$ and that we find a solution which is approximately the same as that reported in [1].

```
fprintf('\nCost function returned by ga = %g\n', fbestDisc);
```

```
Cost function returned by ga = 65226.5
```

Summary

This example illustrates how to use the genetic algorithm solver, `ga`, to solve a constrained nonlinear optimization problem which has integer constraints. The example also shows how to handle problems that have discrete variables in the problem formulation.

References

[1] Survey of discrete variable optimization for structural design, P.B. Thanedar, G.N. Vanderplaats, J. Struct. Eng., 121 (3), 301-306 (1995)

Nonlinear Constraint Solver Algorithms

In this section...

Augmented Lagrangian Genetic Algorithm

By default, the genetic algorithm uses the Augmented Lagrangian Genetic Algorithm (ALGA) to solve nonlinear constraint problems without integer constraints. The optimization problem solved by the ALGA algorithm is

$$\min_x f(x)$$

such that

$$\begin{aligned} c_i(x) &\le 0,\ i = 1 \ldots m \\ ceq_i(x) &= 0,\ i = m+1 \ldots mt \\ A \cdot x &\le b \\ Aeq \cdot x &= beq \\ lb \le x &\le ub, \end{aligned}$$

where $c(x)$ represents the nonlinear inequality constraints, $ceq(x)$ represents the equality constraints, m is the number of nonlinear inequality constraints, and mt is the total number of nonlinear constraints.

The Augmented Lagrangian Genetic Algorithm (ALGA) attempts to solve a nonlinear optimization problem with nonlinear constraints, linear constraints, and bounds. In this approach, bounds and linear constraints are handled separately from nonlinear constraints. A subproblem is formulated by combining the fitness function and nonlinear constraint function using the Lagrangian and the penalty parameters. A sequence of such optimization problems are approximately minimized using the genetic algorithm such that the linear constraints and bounds are satisfied.

A subproblem formulation is defined as

$$\Theta(x,\lambda,s,\rho) = f(x) - \sum_{i=1}^{m} \lambda_i s_i \log(s_i - c_i(x)) + \sum_{i=m+1}^{mt} \lambda_i ceq_i(x) + \frac{\rho}{2} \sum_{i=m+1}^{mt} ceq_i(x)^2,$$

where

- The components λ_i of the vector λ are nonnegative and are known as Lagrange multiplier estimates
- The elements s_i of the vector s are nonnegative shifts
- ρ is the positive penalty parameter.

The algorithm begins by using an initial value for the penalty parameter (`InitialPenalty`).

The genetic algorithm minimizes a sequence of subproblems, each of which is an approximation of the original problem. Each subproblem has a fixed value of λ, s, and ρ. When the subproblem is minimized to a required accuracy and satisfies feasibility conditions, the Lagrangian estimates are updated. Otherwise, the penalty parameter is increased by a penalty factor (`PenaltyFactor`). This results in a new subproblem formulation and minimization problem. These steps are repeated until the stopping criteria are met.

Each subproblem solution represents one generation. The number of function evaluations per generation is therefore much higher when using nonlinear constraints than otherwise.

Choose the Augmented Lagrangian algorithm by setting the `NonlinConAlgorithm` option to `'auglag'` using `gaoptimset`.

For a complete description of the algorithm, see the following references:

References

[1] Conn, A. R., N. I. M. Gould, and Ph. L. Toint. “A Globally Convergent Augmented Lagrangian Algorithm for Optimization with General Constraints and Simple Bounds,” *SIAM Journal on Numerical Analysis*, Volume 28, Number 2, pages 545–572, 1991.

[2] Conn, A. R., N. I. M. Gould, and Ph. L. Toint. “A Globally Convergent Augmented Lagrangian Barrier Algorithm for Optimization with General Inequality

Constraints and Simple Bounds," *Mathematics of Computation*, Volume 66, Number 217, pages 261–288, 1997.

Penalty Algorithm

The penalty algorithm is similar to the "Integer ga Algorithm" on page 5-35. In its evaluation of the fitness of an individual, `ga` computes a penalty value as follows:

- If the individual is feasible, the penalty function is the fitness function.
- If the individual is infeasible, the penalty function is the maximum fitness function among feasible members of the population, plus a sum of the constraint violations of the (infeasible) individual.

For details of the penalty function, see Deb [1].

Choose the penalty algorithm by setting the `NonlinConAlgorithm` option to `'penalty'` using `gaoptimset`. When you make this choice, `ga` solves the constrained optimization problem as follows.

1. `ga` defaults to the `@gacreationnonlinearfeasible` creation function. This function attempts to create a feasible population with respect to all constraints. `ga` creates enough individuals to match the `PopulationSize` option. For details, see "Penalty Algorithm" on page 10-47.
2. `ga` overrides your choice of selection function, and uses `@selectiontournament` with two individuals per tournament.
3. `ga` proceeds according to "How the Genetic Algorithm Works" on page 5-18, using the penalty function as the fitness measure.

References

[1] Deb, Kalyanmoy. *An efficient constraint handling method for genetic algorithms.* Computer Methods in Applied Mechanics and Engineering, 186(2–4), pp. 311–338, 2000.

Create Custom Plot Function

In this section...

About Custom Plot Functions

If none of the plot functions that come with the software is suitable for the output you want to plot, you can write your own custom plot function, which the genetic algorithm calls at each generation to create the plot. This example shows how to create a plot function that displays the change in the best fitness value from the previous generation to the current generation.

Creating the Custom Plot Function

To create the plot function for this example, copy and paste the following code into a new file in the MATLAB Editor.

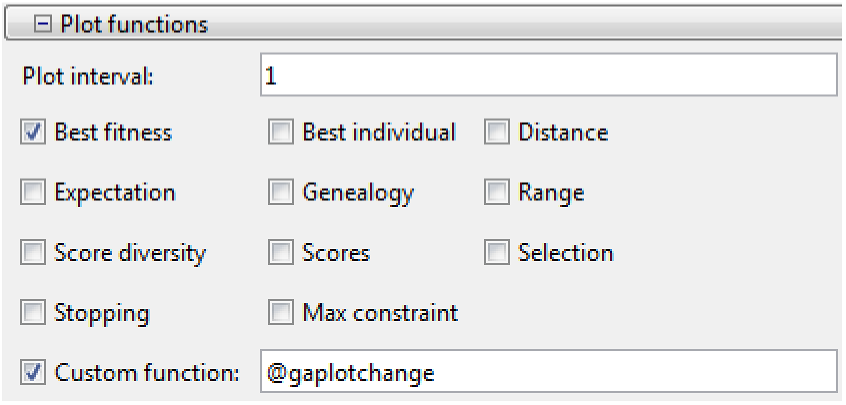
```
function state = gaplotchange(options, state, flag)
% GAPLOTCHANGE Plots the logarithmic change in the best score from the
% previous generation.
%
persistent last_best % Best score in the previous generation

if(strcmp(flag,'init')) % Set up the plot
    xlim([1,options.Generations]);
    axx = gca;
    axx.Yscale = 'log';
    hold on;
    xlabel Generation
    title('Log Absolute Change in Best Fitness Value')
end

best = min(state.Score); % Best score in the current generation
if state.Generation == 0 % Set last_best to best.
```

```
    last_best = best;
else
 change = last_best - best; % Change in best score
 last_best = best;
    if change > 0 % Plot only when the fitness improves
        plot(state.Generation,change,'xr');
    end
end
```

Then save the file as `gaplotchange.m` in a folder on the MATLAB path.

Using the Plot Function

To use the custom plot function, select **Custom** in the **Plot functions** pane and enter `@gaplotchange` in the field to the right. To compare the custom plot with the best fitness value plot, also select **Best fitness**.

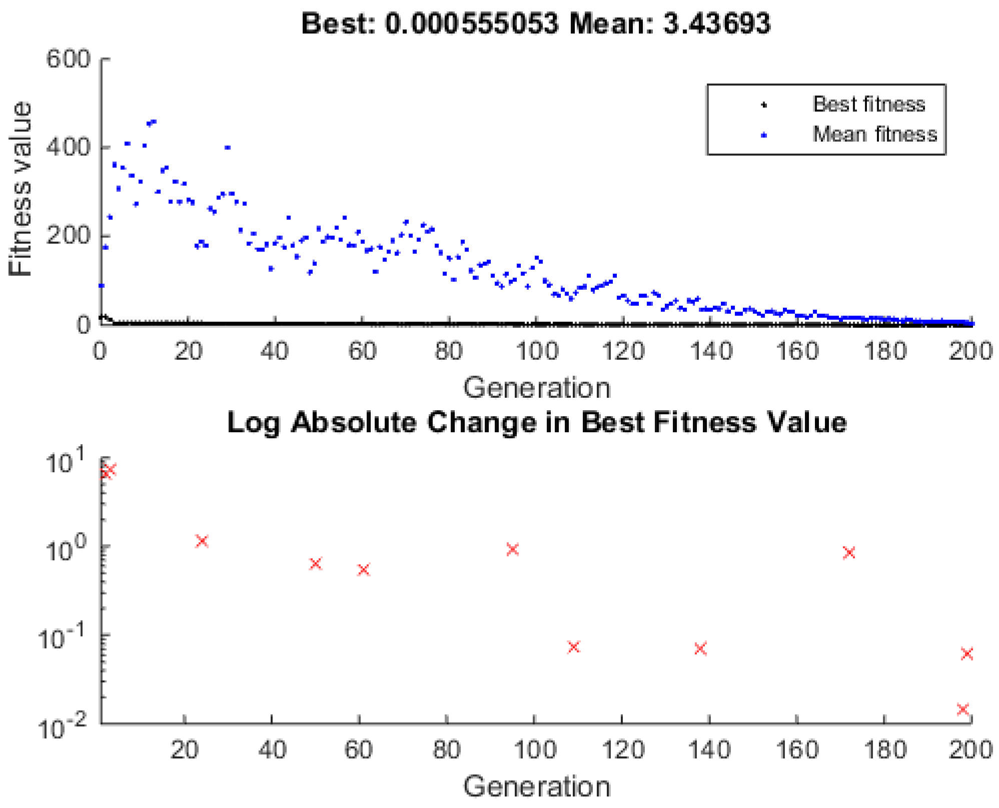

Plot functions

Plot interval: 1

[✓] Best fitness	[ ] Best individual	[ ] Distance
[ ] Expectation	[ ] Genealogy	[ ] Range
[ ] Score diversity	[ ] Scores	[ ] Selection
[ ] Stopping	[ ] Max constraint	

[✓] Custom function: @gaplotchange

Now, if you run the example described in “Minimize Rastrigin's Function” on page 5-6, the tool displays plots similar to those shown in the following figure.



The plot only shows changes that are greater than 0, which are improvements in best fitness. The logarithmic scale enables you to see small changes in the best fitness function that the upper plot does not reveal.

How the Plot Function Works

The plot function uses information contained in the following structures, which the genetic algorithm passes to the function as input arguments:

- `options` — The current options settings
- `state` — Information about the current generation
- `flag` — String indicating the current status of the algorithm

The most important lines of the plot function are the following:

- `persistent last_best`

  Creates the persistent variable `last_best`—the best score in the previous generation. Persistent variables are preserved over multiple calls to the plot function.
- `xlim([1,options.Generations]);`

  `axx = gca;`

  `axx.Yscale = 'log';`

  Sets up the plot before the algorithm starts. `options.Generations` is the maximum number of generations.
- `best = min(state.Score)`

  The field `state.Score` contains the scores of all individuals in the current population. The variable `best` is the minimum score. For a complete description of the fields of the structure state, see “Structure of the Plot Functions” on page 10-31.
- `change = last_best - best`

  The variable change is the best score at the previous generation minus the best score in the current generation.
- `if change > 0`

  Plot only if there is a change in the best fitness.
- `plot(state.Generation,change,'xr')`

  Plots the change at the current generation, whose number is contained in `state.Generation`.

The code for `gaplotchange` contains many of the same elements as the code for `gaplotbestf`, the function that creates the best fitness plot.

Reproduce Results in Optimization App

To reproduce the results of the last run of the genetic algorithm, select the **Use random states from previous run** check box. This resets the states of the random number generators used by the algorithm to their previous values. If you do not change any other settings in the Optimization app, the next time you run the genetic algorithm, it returns the same results as the previous run.

Normally, you should leave **Use random states from previous run** unselected to get the benefit of randomness in the genetic algorithm. Select the **Use random states from previous run** check box if you want to analyze the results of that particular run or show the exact results to others. After the algorithm has run, you can clear your results using the **Clear Status** button in the **Run solver** settings.

Note: If you select **Include information needed to resume this run**, then selecting **Use random states from previous run** has no effect on the initial population created when you import the problem and run the genetic algorithm on it. The latter option is only intended to reproduce results from the beginning of a new run, not from a resumed run.

Resume ga

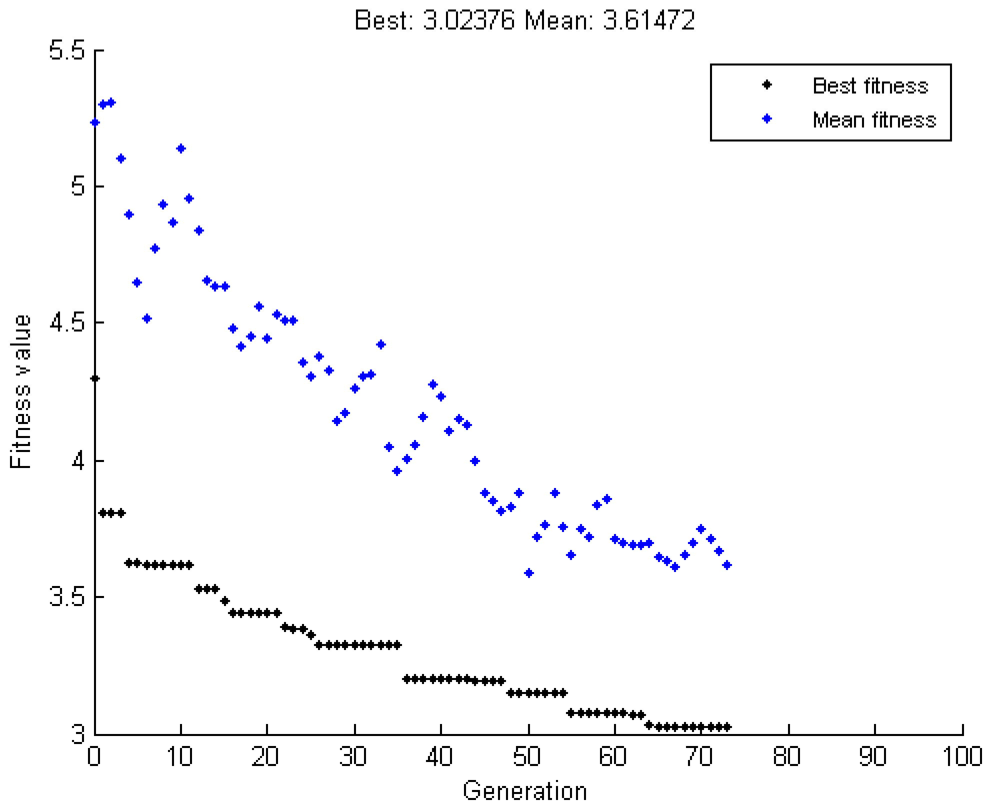
In this section...

Resuming ga From the Final Population

The following example shows how export a problem so that when you import it and click **Start**, the genetic algorithm resumes from the final population saved with the exported problem. To run the example, enter the following in the Optimization app:

1. Set **Fitness function** to `@ackleyfcn`, which computes Ackley's function, a test function provided with the software.
2. Set **Number of variables** to `10`.
3. Select **Best fitness** in the **Plot functions** pane.
4. Click **Start**.

This displays the following plot, or similar.



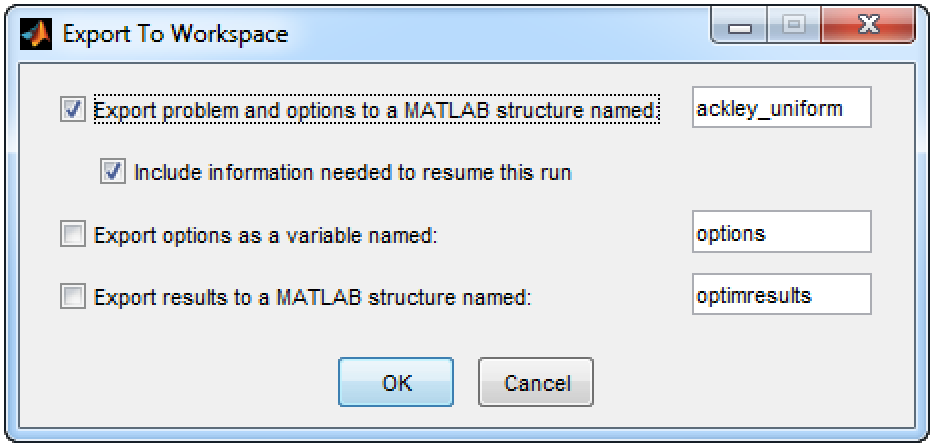
Suppose you want to experiment by running the genetic algorithm with other options settings, and then later restart this run from its final population with its current options settings. You can do this using the following steps:

1 Click **Export to Workspace**.

2 In the dialog box that appears,

- Select **Export problem and options to a MATLAB structure named**.
- Enter a name for the problem and options, such as `ackley_uniform`, in the text field.
- Select **Include information needed to resume this run**.

The dialog box should now appear as in the following figure.



[✓] Export problem and options to a MATLAB structure named: ackley_uniform

[✓] Include information needed to resume this run

[ ] Export options as a variable named: options

[ ] Export results to a MATLAB structure named: optimresults

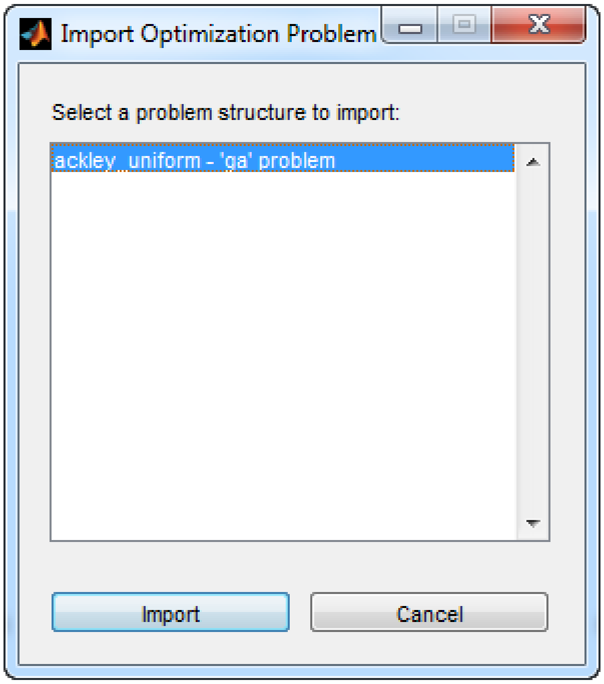
3 Click **OK**.

This exports the problem and options to a structure in the MATLAB workspace. You can view the structure in the MATLAB Command Window by entering

```
ackley_uniform

ackley_uniform =

    fitnessfcn: @ackleyfcn
         nvars: 10
         Aineq: []
         bineq: []
           Aeq: []
           beq: []
            lb: []
            ub: []
       nonlcon: []
        intcon: []
      rngstate: []
        solver: 'ga'
       options: [1x1 struct]
```

After running the genetic algorithm with different options settings or even a different fitness function, you can restore the problem as follows:

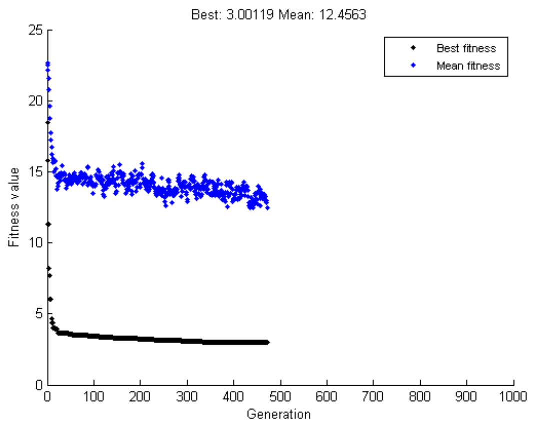
1 Select **Import Problem** from the **File** menu. This opens the dialog box shown in the following figure.

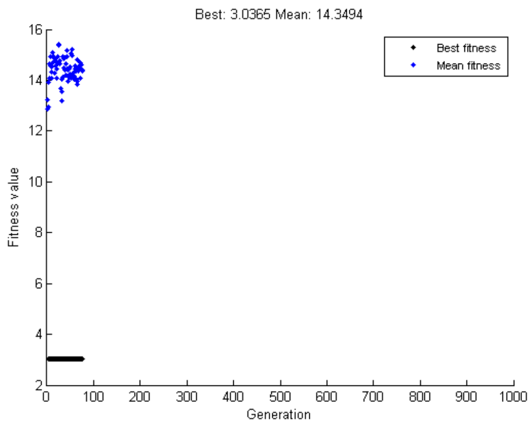

Import Optimization Problem
Select a problem structure to import:
ackley_uniform - 'ga' problem
Import
Cancel

2 Select `ackley_uniform`.

3 Click **Import**.

This sets the **Initial population** and **Initial scores** fields in the **Population** panel to the final population of the run before you exported the problem. All other options are restored to their setting during that run. When you click **Start**, the genetic algorithm resumes from the saved final population. The following figure shows the best fitness plots from the original run and the restarted run.





Note If, after running the genetic algorithm with the imported problem, you want to restore the genetic algorithm's default behavior of generating a random initial population, delete the population in the **Initial population** field.

The version of Ackley's function in the toolbox differs from the published version of Ackley's function in Ackley [1]. The toolbox version has another exponential applied, leading to flatter regions, so a more difficult optimization problem.

References

[1] Ackley, D. H. *A connectionist machine for genetic hillclimbing*. Kluwer Academic Publishers, Boston, 1987.

Resuming ga From a Previous Run

By default, `ga` creates a new initial population each time you run it. However, you might get better results by using the final population from a previous run as the initial population for a new run. To do so, you must have saved the final population from the previous run by calling `ga` with the syntax

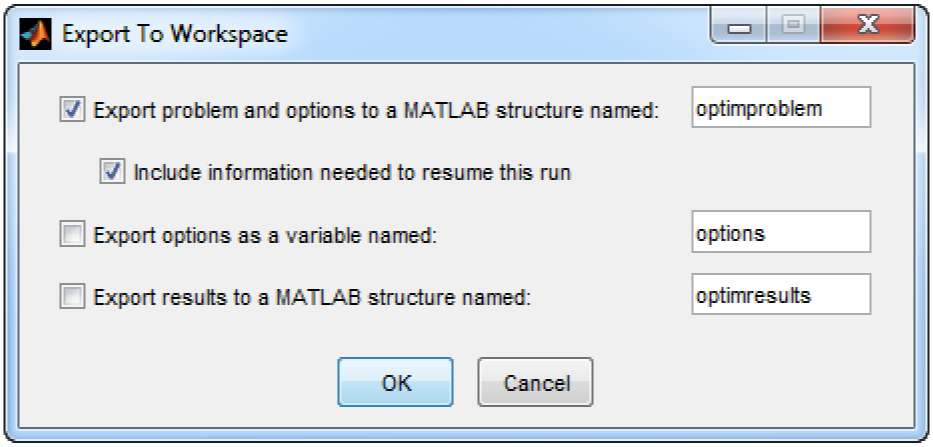
```
[x,fval,exitflag,output,final_pop] = ga(@fitnessfcn, nvars);
```

The last output argument is the final population. To run `ga` using `final_pop` as the initial population, enter

```
options = gaoptimset('InitialPop', final_pop);
[x,fval,exitflag,output,final_pop2] = ...
  ga(@fitnessfcn,nvars,[],[],[],[],[],[],[],options);
```

You can then use `final_pop2`, the final population from the second run, as the initial population for a third run.

In Optimization app, you can choose to export a problem in a way that lets you resume the run. Simply check the box **Include information needed to resume this run** when exporting the problem.



[✓] Export problem and options to a MATLAB structure named: optimproblem

[✓] Include information needed to resume this run

[ ] Export options as a variable named: options

[ ] Export results to a MATLAB structure named: optimresults

This saves the final population, which becomes the initial population when imported.

If you want to run a problem that was saved with the final population, but would rather not use the initial population, simply delete or otherwise change the initial population in the **Options > Population** pane.

Options and Outputs

In this section...

Running ga with the Default Options

To run the genetic algorithm with the default options, call `ga` with the syntax

```
[x fval] = ga(@fitnessfun, nvars)
```

The input arguments to `ga` are

- `@fitnessfun` — A function handle to the file that computes the fitness function. "Compute Objective Functions" on page 2-2 explains how to write this file.
- `nvars` — The number of independent variables for the fitness function.

The output arguments are

- `x` — The final point
- `fval` — The value of the fitness function at `x`

For a description of additional input and output arguments, see the reference page for `ga`.

You can run the example described in "Minimize Rastrigin's Function" on page 5-6 from the command line by entering

```
rng(1,'twister') % for reproducibility
[x fval] = ga(@rastriginsfcn, 2)
```

This returns

```
Optimization terminated:
 average change in the fitness value less than options.TolFun.

x =
   -0.0017   -0.0185
```

```
fval =
    0.0682
```

Setting Options at the Command Line

You can specify any of the options that are available for `ga` by passing an options structure as an input argument to `ga` using the syntax

```
[x fval] = ga(@fitnessfun,nvars,[],[],[],[],[],[],[],options)
```

This syntax does not specify any linear equality, linear inequality, or nonlinear constraints.

You create the options structure using the function `gaoptimset`.

```
options = gaoptimset(@ga)
```

This returns the structure `options` with the default values for its fields.

```
    options = 

            PopulationType: 'doubleVector'
              PopInitRange: []
            PopulationSize: '50 when numberOfVariables <= 5, else 200'
                EliteCount: '0.05*PopulationSize'
         CrossoverFraction: 0.8000
            ParetoFraction: []
        MigrationDirection: 'forward'
         MigrationInterval: 20
         MigrationFraction: 0.2000
               Generations: '100*numberOfVariables'
                 TimeLimit: Inf
              FitnessLimit: -Inf
             StallGenLimit: 50
                 StallTest: 'averageChange'
            StallTimeLimit: Inf
                    TolFun: 1.0000e-06
                    TolCon: 1.0000e-03
         InitialPopulation: []
             InitialScores: []
        NonlinConAlgorithm: 'auglag'
            InitialPenalty: 10
             PenaltyFactor: 100
              PlotInterval: 1
```

```
           CreationFcn: @gacreationuniform
     FitnessScalingFcn: @fitscalingrank
          SelectionFcn: @selectionstochunif
          CrossoverFcn: @crossoverscattered
           MutationFcn: {[@mutationgaussian]  [1]  [1]}
    DistanceMeasureFcn: []
             HybridFcn: []
               Display: 'final'
              PlotFcns: []
            OutputFcns: []
            Vectorized: 'off'
           UseParallel: 0
```

The function `ga` uses these default values if you do not pass in options as an input argument.

The value of each option is stored in a field of the options structure, such as `options.PopulationSize`. You can display any of these values by entering `options` followed by a period and the name of the field. For example, to display the size of the population for the genetic algorithm, enter

```
options.PopulationSize

ans =

50 when numberOfVariables <= 5, else 200
```

To create an options structure with a field value that is different from the default — for example to set `PopulationSize` to `100` instead of its default value `50` — enter

```
options = gaoptimset('PopulationSize',100)
```

This creates the options structure with all values set to their defaults except for `PopulationSize`, which is set to `100`.

If you now enter,

```
ga(@fitnessfun,nvars,[],[],[],[],[],[],[],options)
```

`ga` runs the genetic algorithm with a population size of `100`.

If you subsequently decide to change another field in the options structure, such as setting `PlotFcns` to `@gaplotbestf`, which plots the best fitness function value at each generation, call `gaoptimset` with the syntax

```
options = gaoptimset(options,'PlotFcns',@plotbestf)
```

This preserves the current values of all fields of `options` except for `PlotFcns`, which is changed to `@plotbestf`. Note that if you omit the input argument `options`, `gaoptimset` resets `PopulationSize` to its default value `20`.

You can also set both `PopulationSize` and `PlotFcns` with the single command

```
options = gaoptimset('PopulationSize',100,'PlotFcns',@plotbestf)
```

Additional Output Arguments

To get more information about the performance of the genetic algorithm, you can call `ga` with the syntax

```
[x,fval,exitflag,output,population,scores] = ga(@fitnessfcn, nvars)
```

Besides `x` and `fval`, this function returns the following additional output arguments:

- `exitflag` — Integer value corresponding to the reason the algorithm terminated
- `output` — Structure containing information about the performance of the algorithm at each generation
- `population` — Final population
- `scores` — Final scores

See the `ga` reference page for more information about these arguments.

Use Exported Options and Problems

As an alternative to creating an options structure using `gaoptimset`, you can set the values of options in the Optimization app and then export the options to a structure in the MATLAB workspace, as described in the "Importing and Exporting Your Work" section of the Optimization Toolbox documentation. If you export the default options in the Optimization app, the resulting structure `options` has the same settings as the default structure returned by the command

```
options = gaoptimset(@ga)
```

except that the option `'Display'` defaults to `'off'` in an exported structure, and is `'final'` in the default at the command line.

If you export a problem structure, `ga_problem`, from the Optimization app, you can apply `ga` to it using the syntax

```
[x,fval] = ga(ga_problem)
```

The problem structure contains the following fields:

- `fitnessfcn` — Fitness function
- `nvars` — Number of variables for the problem
- `Aineq` — Matrix for inequality constraints
- `Bineq` — Vector for inequality constraints
- `Aeq` — Matrix for equality constraints
- `Beq` — Vector for equality constraints
- `LB` — Lower bound on `x`
- `UB` — Upper bound on `x`
- `nonlcon` — Nonlinear constraint function
- `options` — Options structure

Reproduce Results

Because the genetic algorithm is stochastic—that is, it makes random choices—you get slightly different results each time you run the genetic algorithm. The algorithm uses the default MATLAB pseudorandom number stream. For more information about random number streams, see `RandStream`. Each time `ga` calls the stream, its state changes. So that the next time `ga` calls the stream, it returns a different random number. This is why the output of `ga` differs each time you run it.

If you need to reproduce your results exactly, you can call `ga` with an output argument that contains the current state of the default stream, and then reset the state to this value before running `ga` again. For example, to reproduce the output of `ga` applied to Rastrigin's function, call `ga` with the syntax

```
rng(1,'twister') % for reproducibility
[x fval exitflag output] = ga(@rastriginsfcn, 2);
```

Suppose the results are

```
x,fval

x =
   -0.0017   -0.0185

fval =
    0.0682
```

The state of the stream is stored in `output.rngstate`:

```
output

output =
    problemtype: 'unconstrained'
       rngstate: [1x1 struct]
    generations: 124
      funccount: 6250
        message: 'Optimization terminated: average change in the fitness value less...
  maxconstraint: []
```

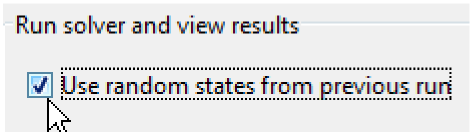
To reset the state, enter

```
stream = RandStream.getGlobalStream;
stream.State = output.rngstate.state;
```

If you now run `ga` a second time, you get the same results as before:

```
[x fval exitflag output] = ga(@rastriginsfcn, 2)

Optimization terminated: average change in the fitness value less than options.TolFun.

x =
   -0.0017   -0.0185

fval =
    0.0682

exitflag =
     1

output =
    problemtype: 'unconstrained'
       rngstate: [1x1 struct]
    generations: 124
      funccount: 6250
        message: 'Optimization terminated: average change in the fitness value less...
  maxconstraint: []
```

You can reproduce your run in the Optimization app by checking the box **Use random states from previous run** in the **Run solver and view results** section.


Run solver and view results
☑ Use random states from previous run

Note If you do not need to reproduce your results, it is better not to set the state of the stream, so that you get the benefit of the randomness in the genetic algorithm.

Run ga from a File

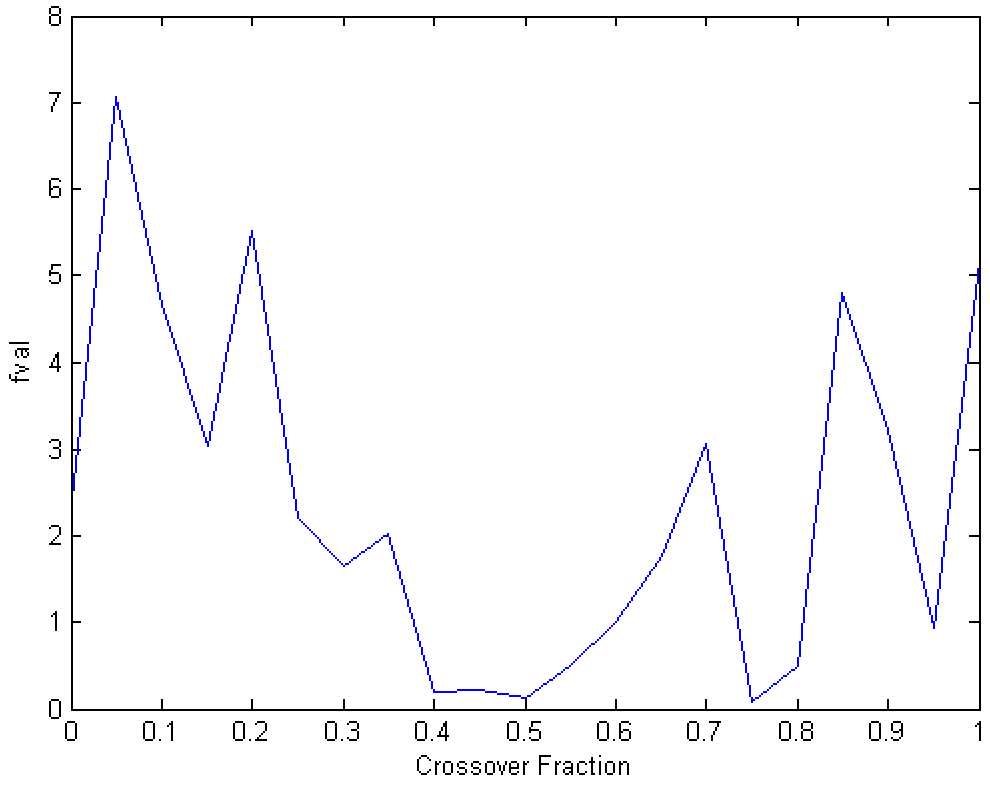
The command-line interface enables you to run the genetic algorithm many times, with different options settings, using a file. For example, you can run the genetic algorithm with different settings for **Crossover fraction** to see which one gives the best results. The following code runs the function `ga` 21 times, varying `options.CrossoverFraction` from `0` to `1` in increments of `0.05`, and records the results.

```
options = gaoptimset('Generations',300,'Display','none');
rng default % for reproducibility
record=[];
for n=0:.05:1
  options = gaoptimset(options,'CrossoverFraction',n);
  [x fval]=ga(@rastriginsfcn,2,[],[],[],[],[],[],[],options);
  record = [record; fval];
end
```

You can plot the values of `fval` against the crossover fraction with the following commands:

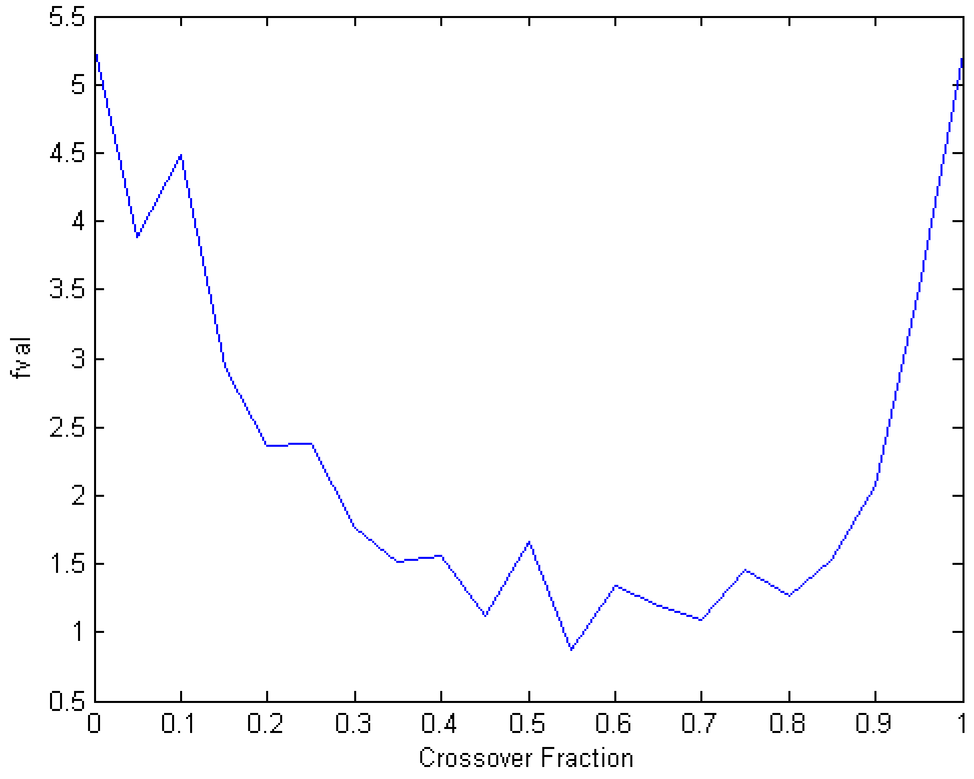
```
plot(0:.05:1, record);
xlabel('Crossover Fraction');
ylabel('fval')
```

The following plot appears.



The plot suggests that you get the best results by setting `options.CrossoverFraction` to a value somewhere between `0.4` and `0.8`.

You can get a smoother plot of `fval` as a function of the crossover fraction by running `ga` 20 times and averaging the values of `fval` for each crossover fraction. The following figure shows the resulting plot.



This plot also suggests the range of best choices for `options.CrossoverFraction` is `0.4` to `0.8`.

Population Diversity

In this section...
"Importance of Population Diversity" on page 5-73
"Setting the Initial Range" on page 5-73
"Linearly Constrained Population and Custom Plot Function" on page 5-77
"Setting the Population Size" on page 5-82

Importance of Population Diversity

One of the most important factors that determines the performance of the genetic algorithm performs is the *diversity* of the population. If the average distance between individuals is large, the diversity is high; if the average distance is small, the diversity is low. Getting the right amount of diversity is a matter of start and error. If the diversity is too high or too low, the genetic algorithm might not perform well.

This section explains how to control diversity by setting the **Initial range** of the population. "Setting the Amount of Mutation" on page 5-88 describes how the amount of mutation affects diversity.

This section also explains how to set the population size.

Setting the Initial Range

By default, `ga` creates a random initial population using a creation function. You can specify the range of the vectors in the initial population in the **Initial range** field in **Population** options.

Note The initial range restricts the range of the points in the *initial* population by specifying the lower and upper bounds. Subsequent generations can contain points whose entries do not lie in the initial range. Set upper and lower bounds for all generations in the **Bounds** fields in the **Constraints** panel.

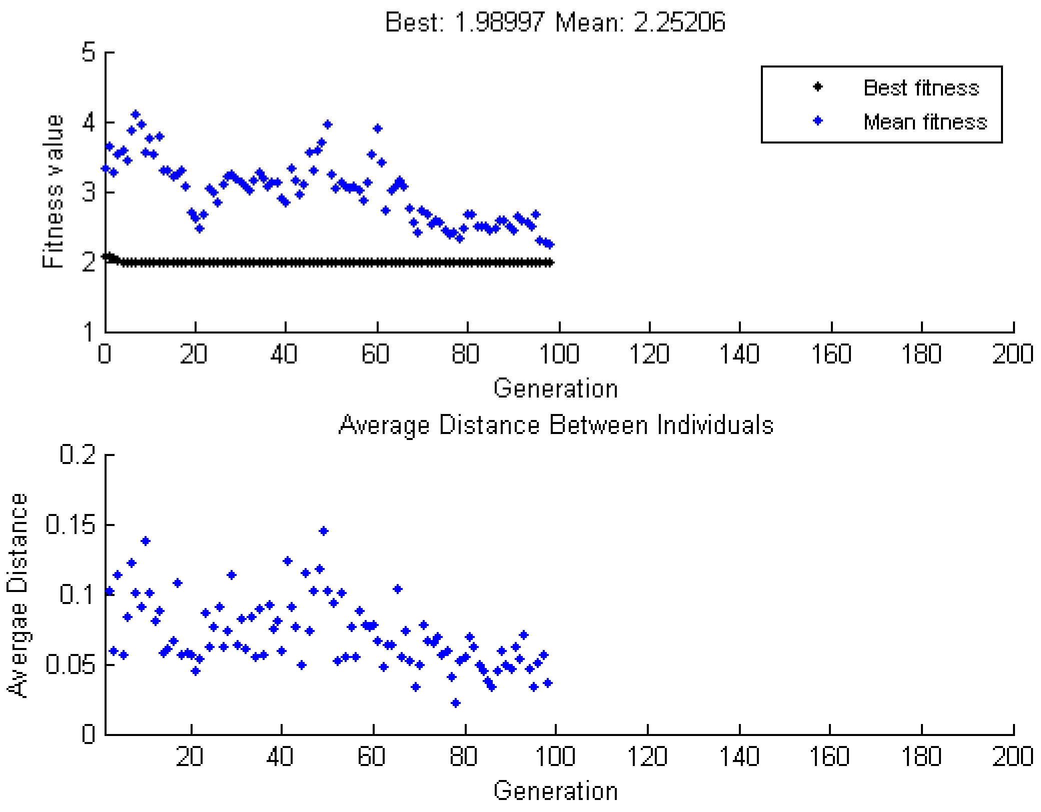
If you know approximately where the solution to a problem lies, specify the initial range so that it contains your guess for the solution. However, the genetic algorithm can find the solution even if it does not lie in the initial range, if the population has enough diversity.

The following example shows how the initial range affects the performance of the genetic algorithm. The example uses Rastrigin's function, described in “Minimize Rastrigin's Function” on page 5-6. The minimum value of the function is 0, which occurs at the origin.

To run the example, open the `ga` solver in the Optimization app by entering `optimtool('ga')` at the command line. Set the following:

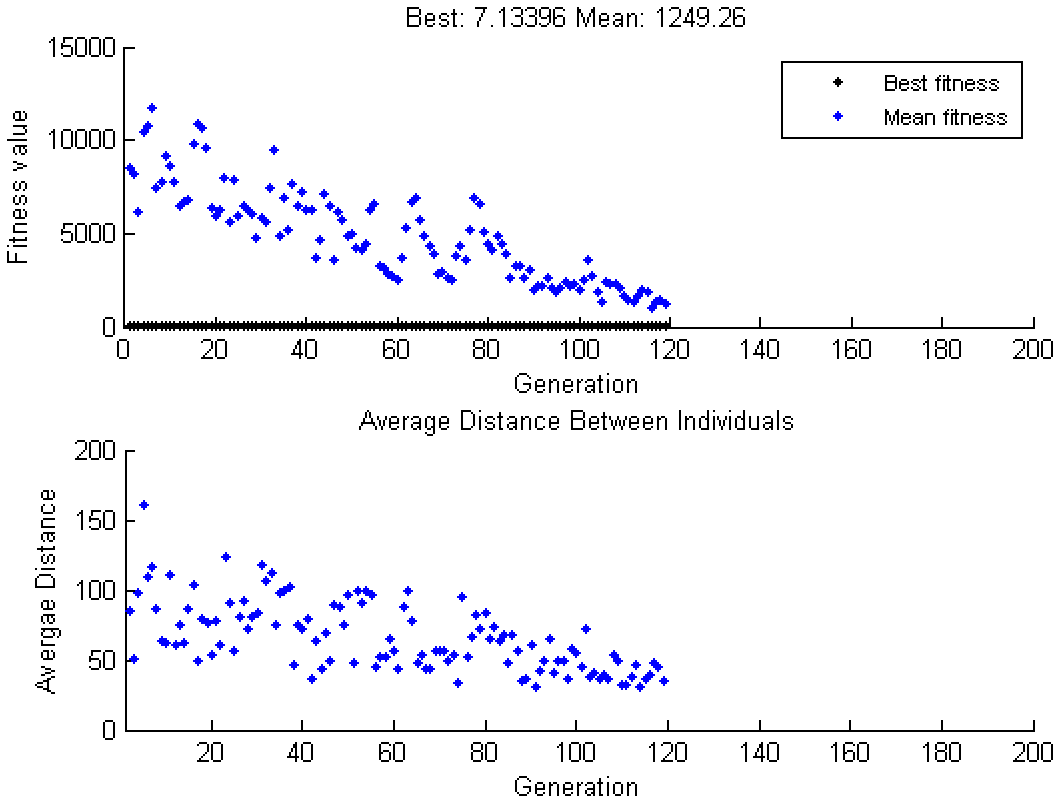
- Set **Fitness function** to `@rastriginsfcn`.
- Set **Number of variables** to `2`.
- Select **Best fitness** in the **Plot functions** pane of the **Options** pane.
- Select **Distance** in the **Plot functions** pane.
- Set **Initial range** in the **Population** pane of the **Options** pane to `[1;1.1]`.

Click **Start** in **Run solver and view results**. Although the results of genetic algorithm computations are random, your results are similar to the following figure, with a best fitness function value of approximately 2.



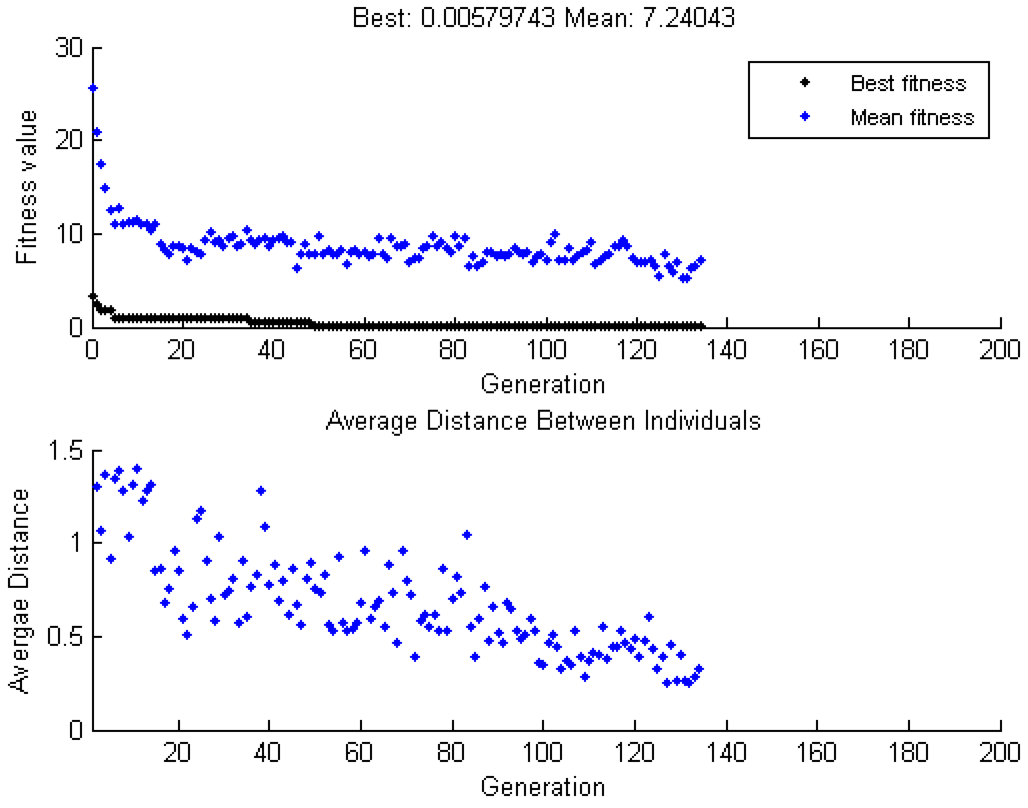
The upper plot, which displays the best fitness at each generation, shows little progress in lowering the fitness value. The lower plot shows the average distance between individuals at each generation, which is a good measure of the diversity of a population. For this setting of initial range, there is too little diversity for the algorithm to make progress.

Next, try setting **Initial range** to `[1;100]` and running the algorithm. This time the results are more variable. You might obtain a plot with a best fitness value of about 7, as in the following plot. You might obtain different results.



This time, the genetic algorithm makes progress, but because the average distance between individuals is so large, the best individuals are far from the optimal solution.

Finally, set **Initial range** to `[1;2]` and run the genetic algorithm. Again, there is variability in the result, but you might obtain a result similar to the following figure. Run the optimization several times, and you eventually obtain a final point near `[0;0]`, with a fitness function value near `0`.



The diversity in this case is better suited to the problem, so `ga` usually returns a better result than in the previous two cases.

Linearly Constrained Population and Custom Plot Function

This example shows how the default creation function for linearly constrained problems, `gacreationlinearfeasible`, creates a well-dispersed population that satisfies linear constraints and bounds. It also contains an example of a custom plot function.

The problem uses the objective function in `lincontest6.m`, a quadratic:

$$f(x) = \frac{x_1^2}{2} + x_2^2 - x_1 x_2 - 2x_1 - 6x_2.$$

To see code for the function, enter `type lincontest6` at the command line. The constraints are three linear inequalities:
$x_1 + x_2 \leq 2$,
$-x_1 + 2x_2 \leq 2$,
$2x_1 + x_2 \leq 3$.

Also, the variables x_i are restricted to be positive.

1 Create a custom plot function file by cutting and pasting the following code into a new function file in the MATLAB Editor:

```
function state = gaplotshowpopulation2(unused,state,flag,fcn)
% This plot function works in 2-d only
if size(state.Population,2) > 2
    return;
end
if nargin < 4 % check to see if fitness function exists
    fcn = [];
end
% Dimensions to plot
dimensionsToPlot = [1 2];

switch flag
    % Plot initialization
    case 'init'
        pop = state.Population(:,dimensionsToPlot);
        plotHandle = plot(pop(:,1),pop(:,2),'*');
        set(plotHandle,'Tag','gaplotshowpopulation2')
        title('Population plot in two dimension',...
                    'interp','none')
        xlabelStr = sprintf('%s %s','Variable ',...
                    num2str(dimensionsToPlot(1)));
        ylabelStr = sprintf('%s %s','Variable ',...
                    num2str(dimensionsToPlot(2)));
        xlabel(xlabelStr,'interp','none');
        ylabel(ylabelStr,'interp','none');
        hold on;

        % plot the inequalities
        plot([0 1.5],[2 0.5],'m-.') %  x1 + x2 <= 2
        plot([0 1.5],[1 3.5/2],'m-.'); % -x1 + 2*x2 <= 2
        plot([0 1.5],[3 0],'m-.'); % 2*x1 + x2 <= 3
        % plot lower bounds
        plot([0 0], [0 2],'m-.'); % lb = [ 0 0];
        plot([0 1.5], [0 0],'m-.'); % lb = [ 0 0];
        set(gca,'xlim',[-0.7,2.2])
        set(gca,'ylim',[-0.7,2.7])

        % Contour plot the objective function
        if ~isempty(fcn) % if there is a fitness function
            range = [-0.5,2;-0.5,2];
            pts = 100;
            span = diff(range')/(pts - 1);
            x = range(1,1): span(1) : range(1,2);
            y = range(2,1): span(2) : range(2,2);

            pop = zeros(pts * pts,2);
            values = zeros(pts,1);
            k = 1;
            for i = 1:pts
                for j = 1:pts
                    pop(k,:) = [x(i),y(j)];
                    values(k) = fcn(pop(k,:));
                    k = k + 1;
                end
            end
            values = reshape(values,pts,pts);
            contour(x,y,values);
            colorbar
        end
        % Pause for three seconds to view the initial plot
        pause(3);
    case 'iter'
        pop = state.Population(:,dimensionsToPlot);
        plotHandle = findobj(get(gca,'Children'),'Tag',...
                     'gaplotshowpopulation2');
        set(plotHandle,'Xdata',pop(:,1),'Ydata',pop(:,2));
end
```

The custom plot function plots the lines representing the linear inequalities and bound constraints, plots level curves of the fitness function, and plots the population as it evolves. This plot function expects to have not only the usual inputs `(options,state,flag)`, but also a function handle to the fitness function,

`@lincontest6` in this example. To generate level curves, the custom plot function needs the fitness function.

2 At the command line, enter the constraints as a matrix and vectors:

```
A = [1,1;-1,2;2,1]; b = [2;2;3]; lb = zeros(2,1);
```

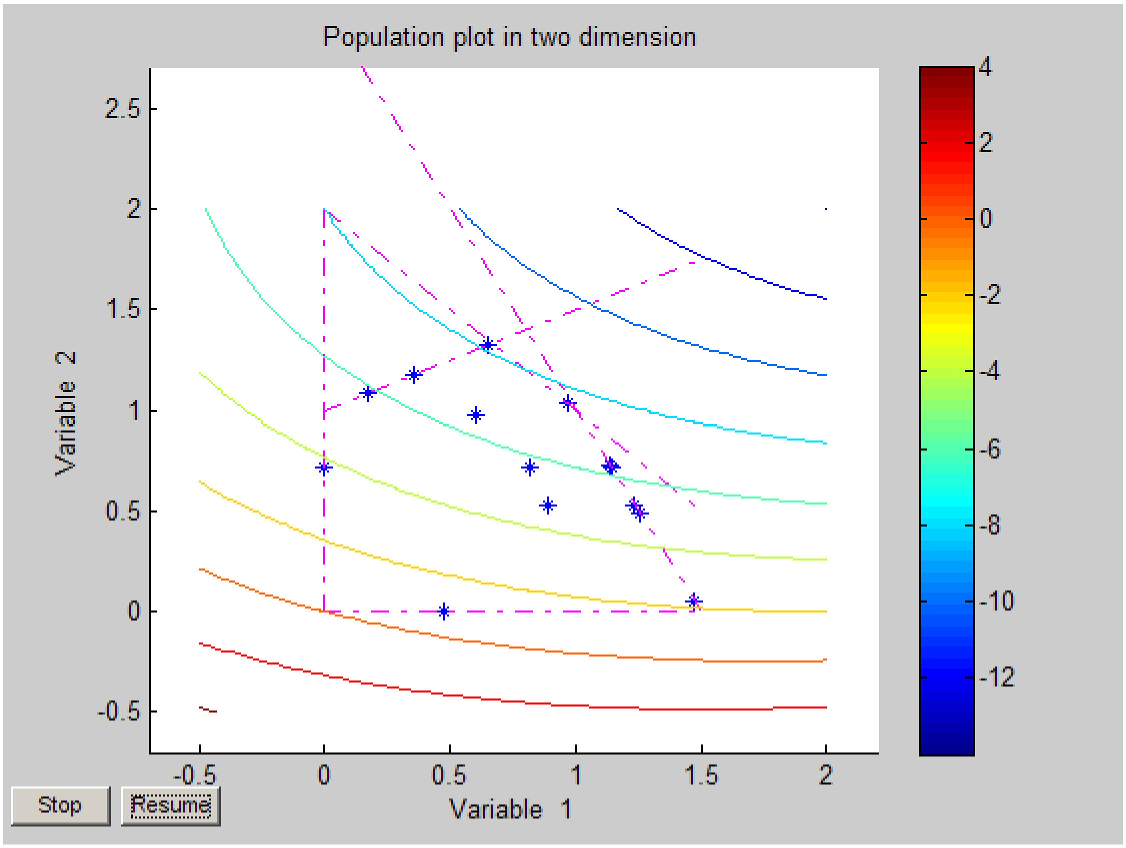
3 Set `options` to use `gaplotshowpopulation2`, and pass in `@lincontest6` as the fitness function handle:

```
options = gaoptimset('PlotFcns',...
          {{@gaplotshowpopulation2,@lincontest6}});
```

4 Run the optimization using `options`:

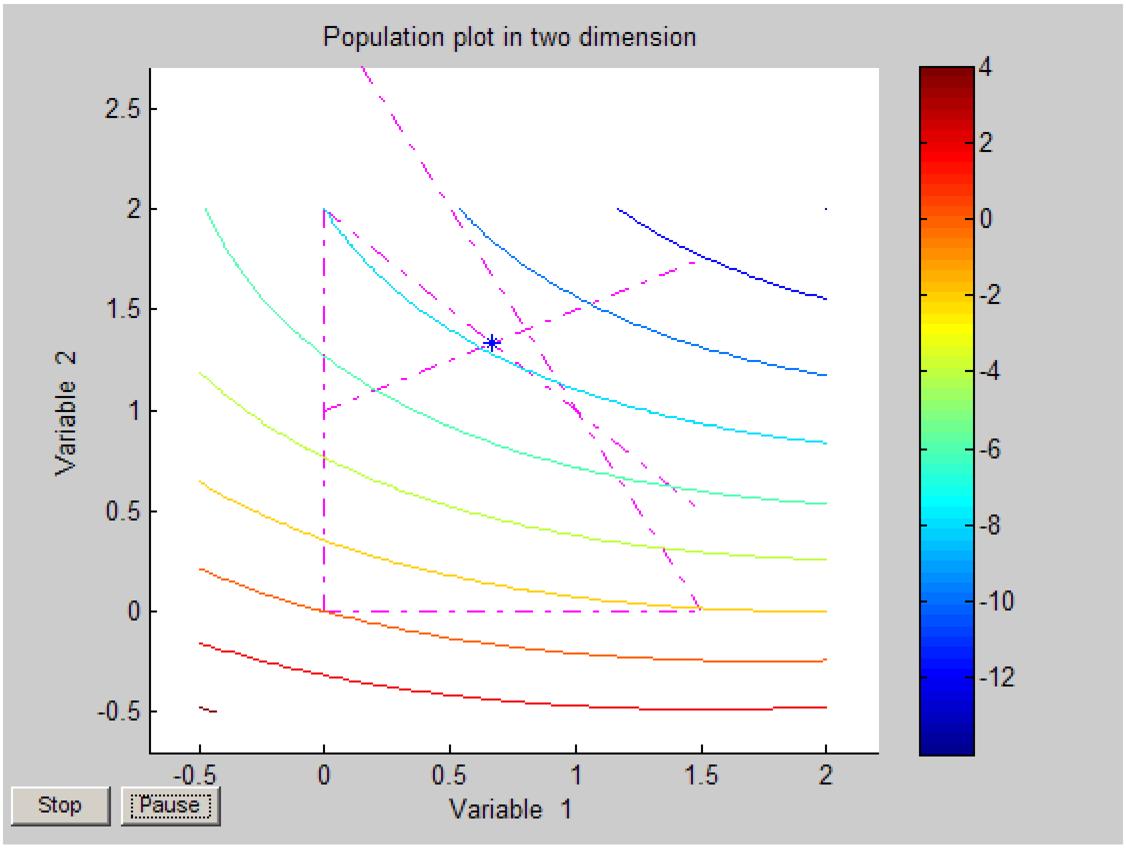
```
[x,fval] = ga(@lincontest6,2,A,b,[],[],lb,[],[],options);
```

A plot window appears showing the linear constraints, bounds, level curves of the objective function, and initial distribution of the population:



You can see that the initial population is biased to lie on the constraints. This bias exists when there are linear constraints.

The population eventually concentrates around the minimum point:



Setting the Population Size

The **Population size** field in **Population** options determines the size of the population at each generation. Increasing the population size enables the genetic algorithm to search more points and thereby obtain a better result. However, the larger the population size, the longer the genetic algorithm takes to compute each generation.

Note You should set **Population size** to be at least the value of **Number of variables**, so that the individuals in each population span the space being searched.

You can experiment with different settings for **Population size** that return good results without taking a prohibitive amount of time to run.

Fitness Scaling

In this section...

Scaling the Fitness Scores

Fitness scaling converts the raw fitness scores that are returned by the fitness function to values in a range that is suitable for the selection function. The selection function uses the scaled fitness values to select the parents of the next generation. The selection function assigns a higher probability of selection to individuals with higher scaled values.

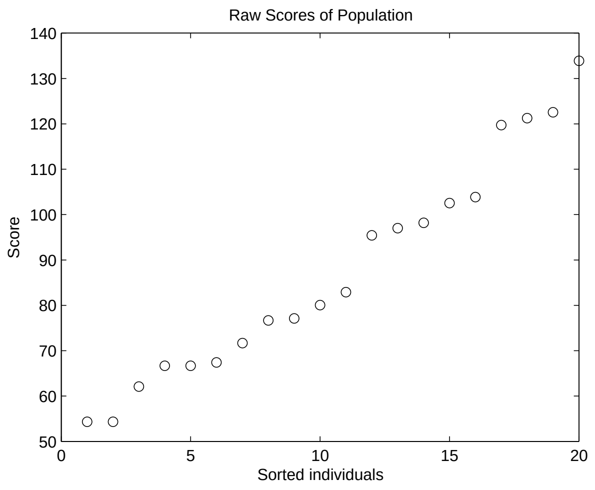
The range of the scaled values affects the performance of the genetic algorithm. If the scaled values vary too widely, the individuals with the highest scaled values reproduce too rapidly, taking over the population gene pool too quickly, and preventing the genetic algorithm from searching other areas of the solution space. On the other hand, if the scaled values vary only a little, all individuals have approximately the same chance of reproduction and the search will progress very slowly.

The default fitness scaling option, `Rank`, scales the raw scores based on the rank of each individual instead of its score. The rank of an individual is its position in the sorted scores: the rank of the most fit individual is 1, the next most fit is 2, and so on. The rank scaling function assigns scaled values so that

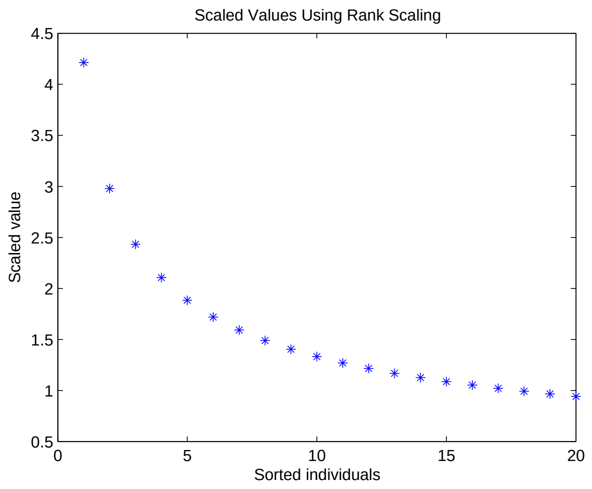
- The scaled value of an individual with rank n is proportional to $1/\sqrt{n}$.
- The sum of the scaled values over the entire population equals the number of parents needed to create the next generation.

Rank fitness scaling removes the effect of the spread of the raw scores.

The following plot shows the raw scores of a typical population of 20 individuals, sorted in increasing order.



The following plot shows the scaled values of the raw scores using rank scaling.

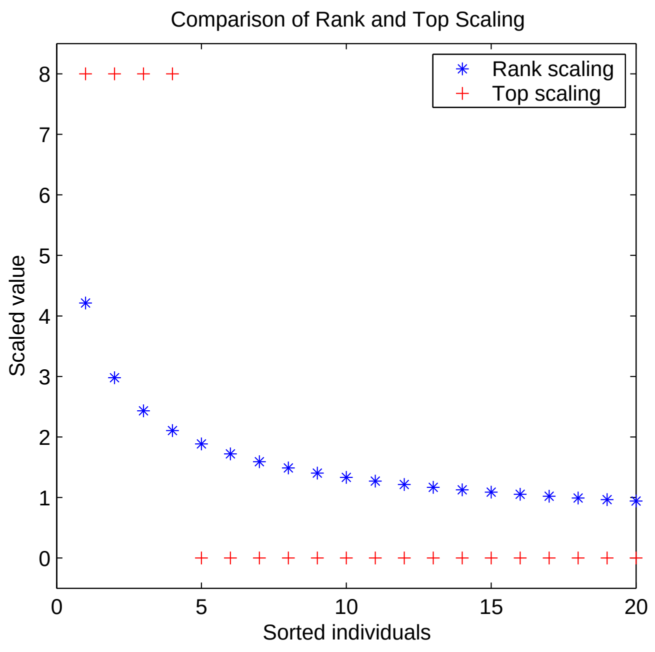


Because the algorithm minimizes the fitness function, lower raw scores have higher scaled values. Also, because rank scaling assigns values that depend only on an individual's rank, the scaled values shown would be the same for any population of size 20 and number of parents equal to 32.

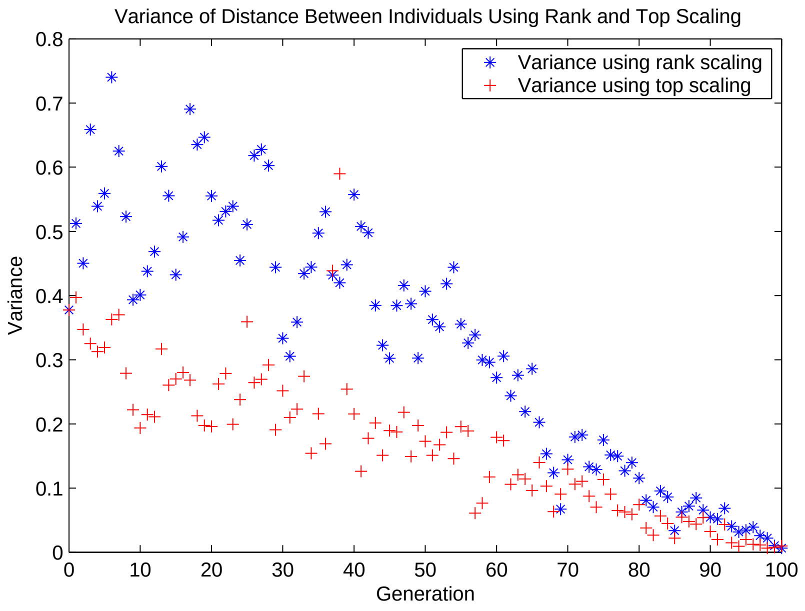
Comparing Rank and Top Scaling

To see the effect of scaling, you can compare the results of the genetic algorithm using rank scaling with one of the other scaling options, such as `Top`. By default, top scaling assigns 40 percent of the fittest individuals to the same scaled value and assigns the rest of the individuals to value 0. Using the default selection function, only 40 percent of the fittest individuals can be selected as parents.

The following figure compares the scaled values of a population of size 20 with number of parents equal to 32 using rank and top scaling.



Because top scaling restricts parents to the fittest individuals, it creates less diverse populations than rank scaling. The following plot compares the variances of distances between individuals at each generation using rank and top scaling.


Variance of Distance Between Individuals Using Rank and Top Scaling
Variance using rank scaling
Variance using top scaling
Variance
Generation
0
0.1
0.2
0.3
0.4
0.5
0.6
0.7
0.8
0
10
20
30
40
50
60
70
80
90
100

Vary Mutation and Crossover

In this section...

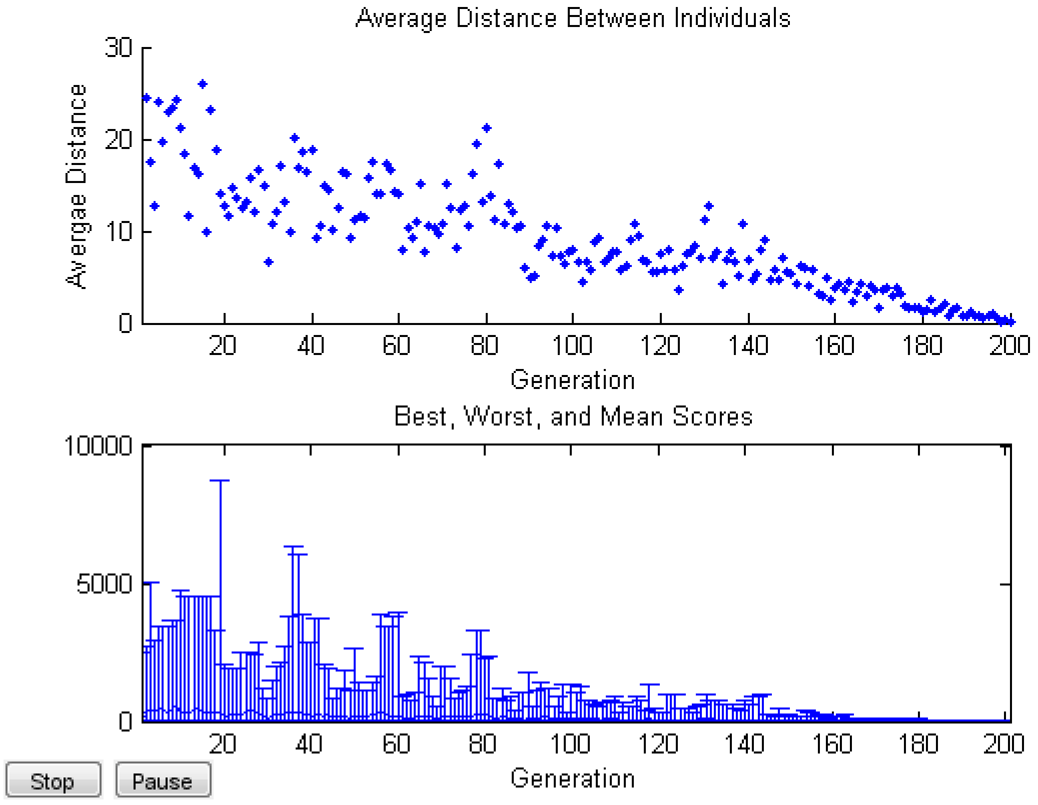
Setting the Amount of Mutation

The genetic algorithm applies mutations using the option that you specify on the **Mutation function** pane. The default mutation option, `Gaussian`, adds a random number, or *mutation*, chosen from a Gaussian distribution, to each entry of the parent vector. Typically, the amount of mutation, which is proportional to the standard deviation of the distribution, decreases at each new generation. You can control the average amount of mutation that the algorithm applies to a parent in each generation through the **Scale** and **Shrink** options:

- **Scale** controls the standard deviation of the mutation at the first generation, which is **Scale** multiplied by the range of the initial population, which you specify by the **Initial range** option.
- **Shrink** controls the rate at which the average amount of mutation decreases. The standard deviation decreases linearly so that its final value equals 1 – **Shrink** times its initial value at the first generation. For example, if **Shrink** has the default value of `1`, then the amount of mutation decreases to 0 at the final step.

You can see the effect of mutation by selecting the plot options **Distance** and **Range**, and then running the genetic algorithm on a problem such as the one described in "Minimize Rastrigin's Function" on page 5-6. The following figure shows the plot after setting the random number generator.

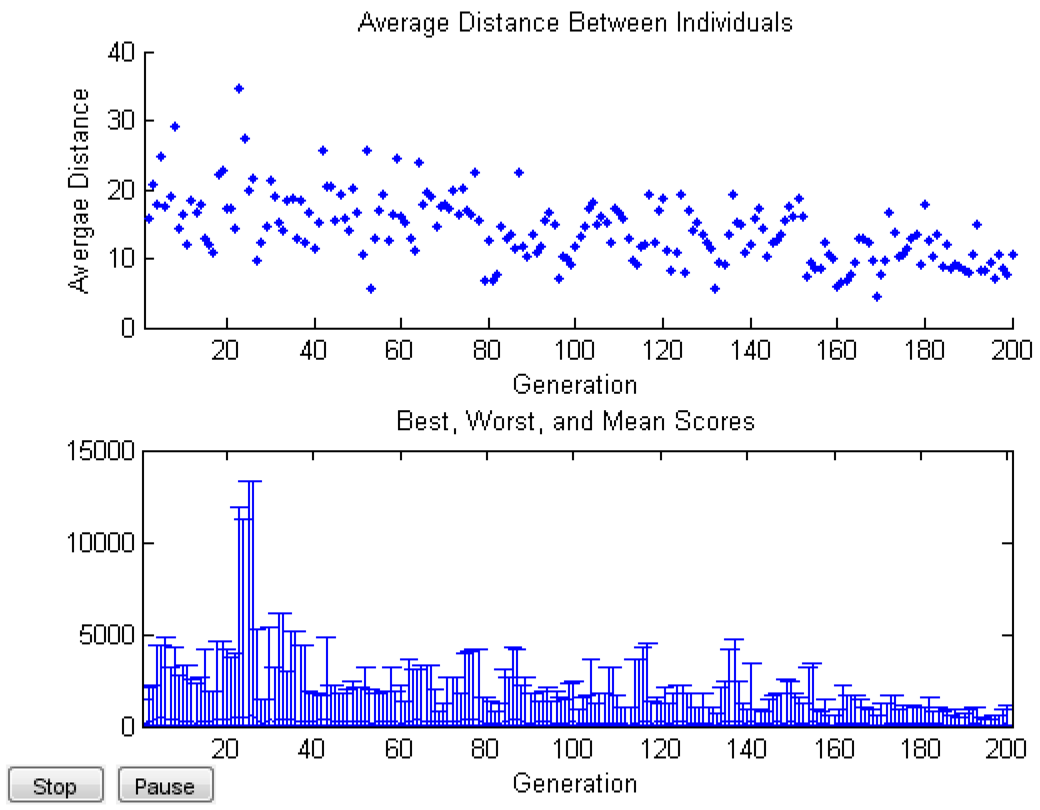
```
rng default % for reproducibility
options = gaoptimset('PlotFcns',{@gaplotdistance,@gaplotrange},...
    'StallGenLimit',200); % to get a long run
[x,fval] = ga(@rastriginsfcn,2,[],[],[],[],[],[],[],options);
```



The upper plot displays the average distance between points in each generation. As the amount of mutation decreases, so does the average distance between individuals, which is approximately 0 at the final generation. The lower plot displays a vertical line at each generation, showing the range from the smallest to the largest fitness value, as well as mean fitness value. As the amount of mutation decreases, so does the range. These plots show that reducing the amount of mutation decreases the diversity of subsequent generations.

For comparison, the following figure shows the plots for **Distance** and **Range** when you set **Shrink** to `0.5`.

```
options = gaoptimset(options,'MutationFcn',{@mutationgaussian,1,.5});
[x,fval] = ga(@rastriginsfcn,2,[],[],[],[],[],[],[],options);
```



With **Shrink** set to `0.5`, the average amount of mutation decreases by a factor of 1/2 by the final generation. As a result, the average distance between individuals decreases less than before.

Setting the Crossover Fraction

The **Crossover fraction** field, in the **Reproduction** options, specifies the fraction of each population, other than elite children, that are made up of crossover children. A crossover fraction of `1` means that all children other than elite individuals are crossover children, while a crossover fraction of `0` means that all children are mutation children. The following example show that neither of these extremes is an effective strategy for optimizing a function.

The example uses the fitness function whose value at a point is the sum of the absolute values of the coordinates at the points. That is,

$$f(x_1, x_2, \ldots, x_n) = |x_1| + |x_2| + \cdots + |x_n|.$$

You can define this function as an anonymous function by setting **Fitness function** to

```
@(x) sum(abs(x))
```

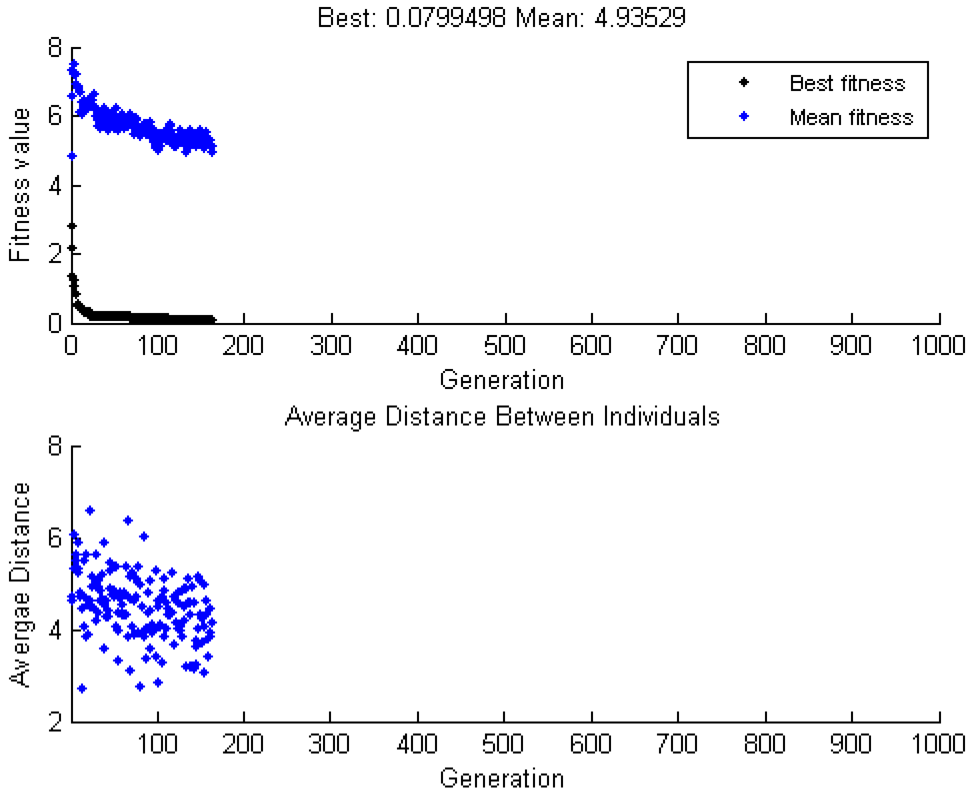
To run the example,

- Set **Fitness function** to `@(x) sum(abs(x))`.
- Set **Number of variables** to `10`.
- Set **Initial range** to `[-1; 1]`.
- Select **Best fitness** and **Distance** in the **Plot functions** pane.

Run the example with the default value of `0.8` for **Crossover fraction**, in the **Options > Reproduction** pane. For reproducibility, switch to the command line and enter

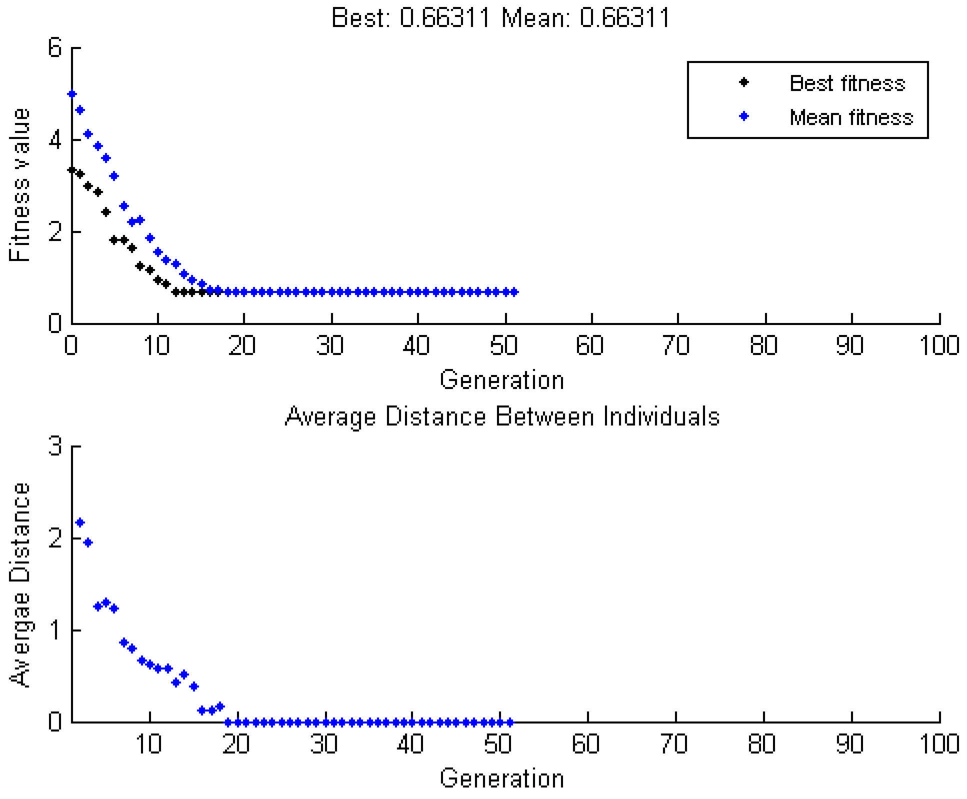
```
rng(14,'twister')
```

Switch back to Optimization app, and click **Run solver and view results > Start**. This returns the best fitness value of approximately `0.0799` and displays the following plots.



Crossover Without Mutation

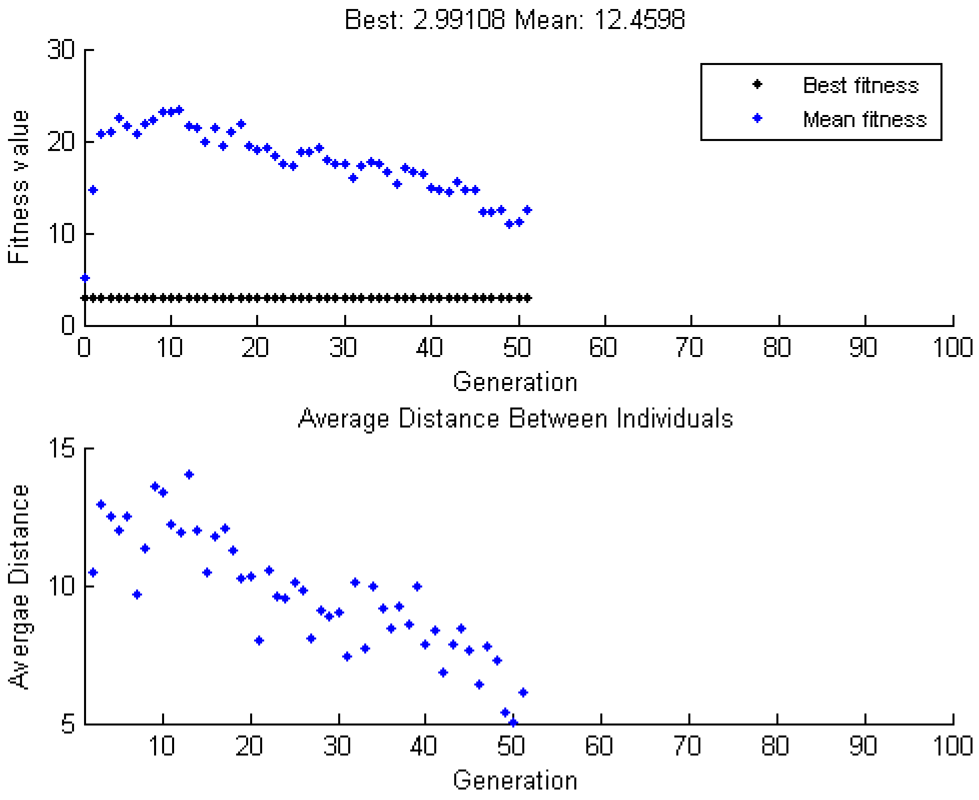
To see how the genetic algorithm performs when there is no mutation, set **Crossover fraction** to `1.0` and click **Start**. This returns the best fitness value of approximately `.66` and displays the following plots.



In this case, the algorithm selects genes from the individuals in the initial population and recombines them. The algorithm cannot create any new genes because there is no mutation. The algorithm generates the best individual that it can using these genes at generation number 8, where the best fitness plot becomes level. After this, it creates new copies of the best individual, which are then are selected for the next generation. By generation number 17, all individuals in the population are the same, namely, the best individual. When this occurs, the average distance between individuals is 0. Since the algorithm cannot improve the best fitness value after generation 8, it stalls after 50 more generations, because **Stall generations** is set to 50.

Mutation Without Crossover

To see how the genetic algorithm performs when there is no crossover, set **Crossover fraction** to 0 and click **Start**. This returns the best fitness value of approximately 3 and displays the following plots.



In this case, the random changes that the algorithm applies never improve the fitness value of the best individual at the first generation. While it improves the individual genes of other individuals, as you can see in the upper plot by the decrease in the mean value of the fitness function, these improved genes are never combined with the genes of the best individual because there is no crossover. As a result, the best fitness plot is level and the algorithm stalls at generation number 50.

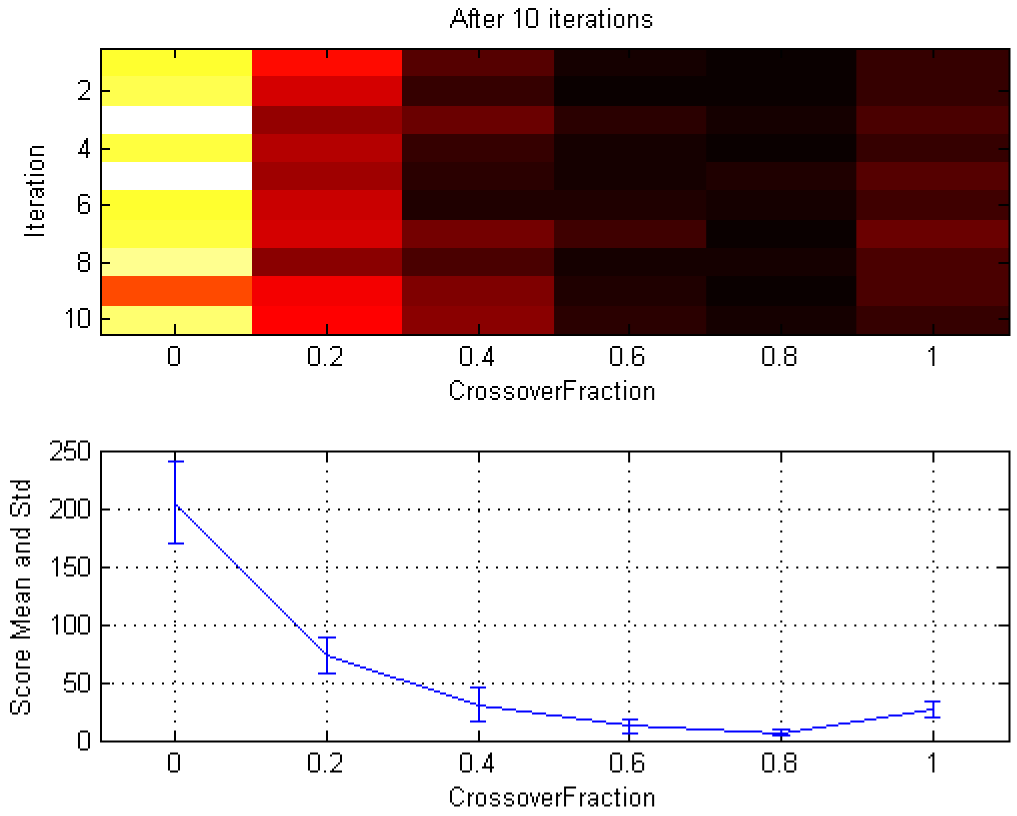
Comparing Results for Varying Crossover Fractions

The example `deterministicstudy.m`, which is included in the software, compares the results of applying the genetic algorithm to Rastrigin's function with **Crossover fraction** set to `0`, `.2`, `.4`, `.6`, `.8`, and `1`. The example runs for 10 generations. At each generation, the example plots the means and standard deviations of the best fitness values in all the preceding generations, for each value of the **Crossover fraction**.

To run the example, enter

```
deterministicstudy
```

at the MATLAB prompt. When the example is finished, the plots appear as in the following figure.



The lower plot shows the means and standard deviations of the best fitness values over 10 generations, for each of the values of the crossover fraction. The upper plot shows a color-coded display of the best fitness values in each generation.

For this fitness function, setting **Crossover fraction** to `0.8` yields the best result. However, for another fitness function, a different setting for **Crossover fraction** might yield the best result.

Global vs. Local Minima Using ga

In this section...
"Searching for a Global Minimum" on page 5-97
"Running the Genetic Algorithm on the Example" on page 5-99

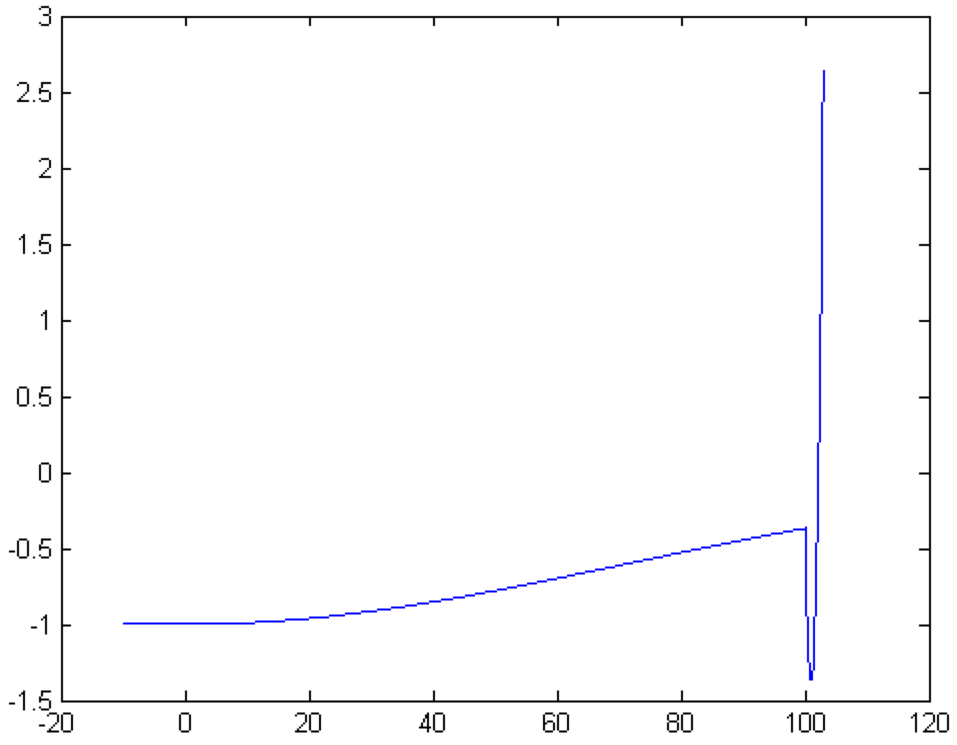
Searching for a Global Minimum

Sometimes the goal of an optimization is to find the global minimum or maximum of a function—a point where the function value is smaller or larger at any other point in the search space. However, optimization algorithms sometimes return a local minimum—a point where the function value is smaller than at nearby points, but possibly greater than at a distant point in the search space. The genetic algorithm can sometimes overcome this deficiency with the right settings.

As an example, consider the following function

$$f(x) = \begin{cases} -\exp\left(-\left(\dfrac{x}{100}\right)^2\right) & \text{for } x \le 100, \\ -\exp(-1) + (x-100)(x-102) & \text{for } x > 100. \end{cases}$$

The following figure shows a plot of the function.



Code for generating the figure

```
t = -10:.1:103;
for ii = 1:length(t)
    y(ii) = two_min(t(ii));
end
plot(t,y)
```

The function has two local minima, one at $x = 0$, where the function value is –1, and the other at $x = 21$, where the function value is $-1 - 1/e$. Since the latter value is smaller, the global minimum occurs at $x = 21$.

Running the Genetic Algorithm on the Example

To run the genetic algorithm on this example,

1 Copy and paste the following code into a new file in the MATLAB Editor.

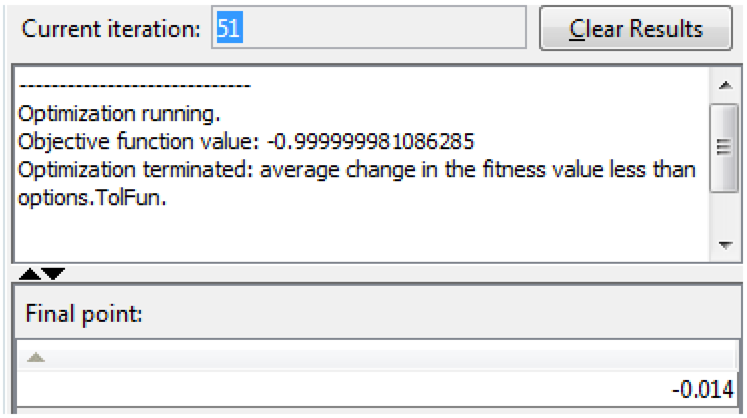
```
function y = two_min(x)
if x <= 100
    y = -exp(-(x/100).^2);
else
    y = -exp(-1) + (x-100)*(x-102);
end
```

2 Save the file as `two_min.m` in a folder on the MATLAB path.

3 In the Optimization app,

- Set **Fitness function** to `@two_min`.
- Set **Number of variables** to `1`.
- Click **Start**.

The genetic algorithm returns a point very close to the local minimum at $x = 0$.

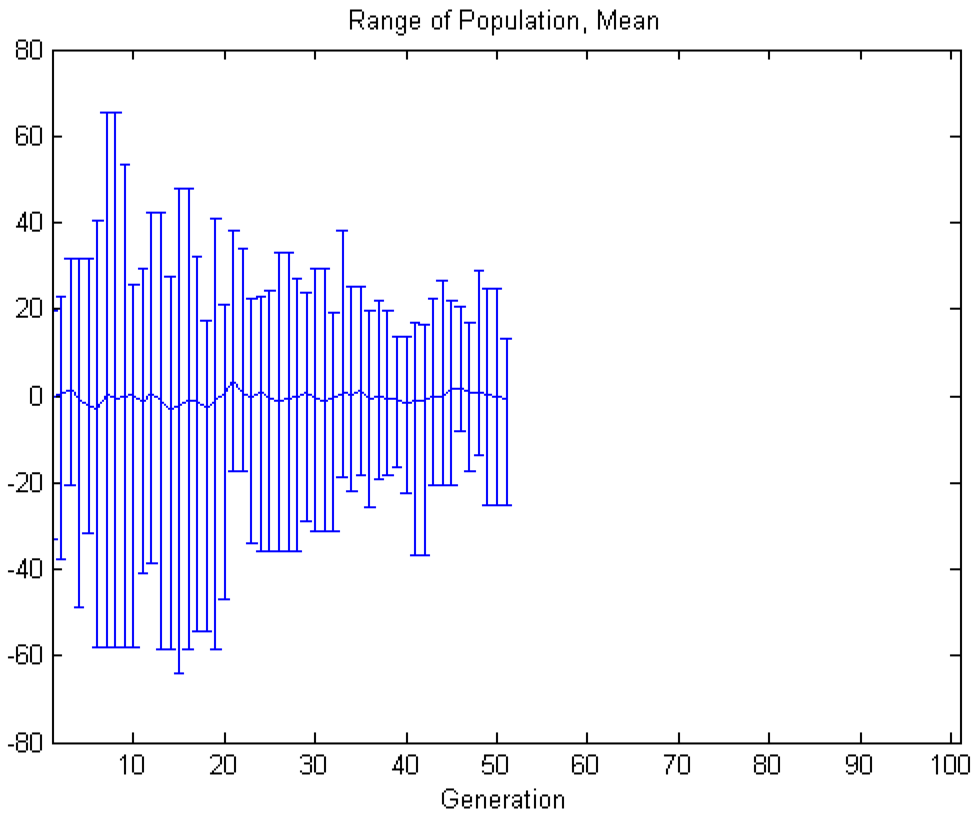


```
------------------------------
Optimization running.
Objective function value: -0.999999981086285
Optimization terminated: average change in the fitness value less than
options.TolFun.
```

Final point:

-0.014

The following custom plot shows why the algorithm finds the local minimum rather than the global minimum. The plot shows the range of individuals in each generation and the population mean.



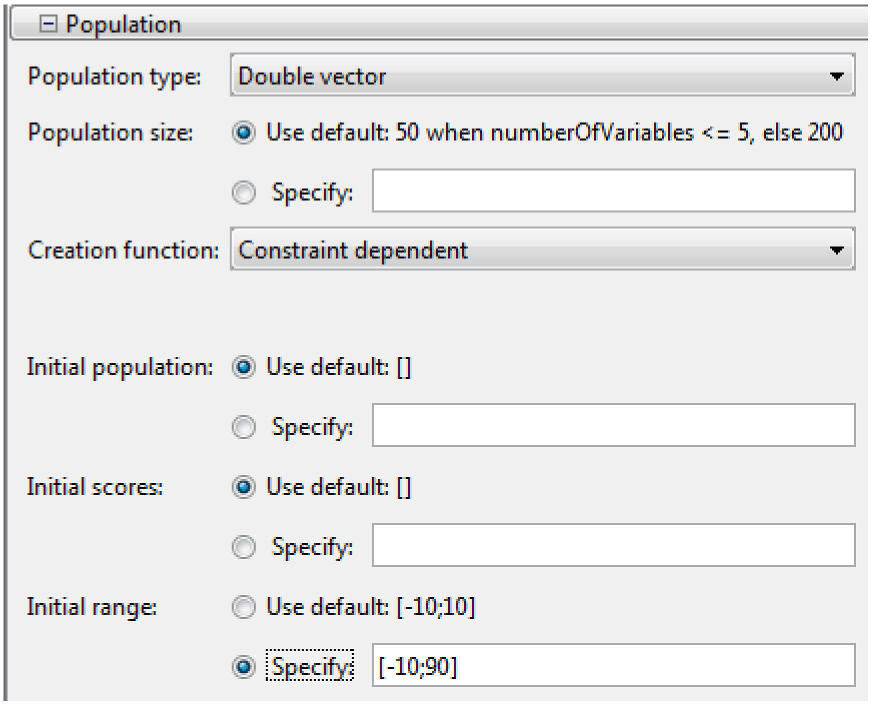
Code for Creating the Figure

```
function state = gaplot1drange(options,state,flag)
%gaplot1drange Plots the mean and the range of the population.
%   STATE = gaplot1drange(OPTIONS,STATE,FLAG) plots the mean and the range
%   (highest and the lowest) of individuals (1-D only).
%
%   Example:
%   Create an options structure that uses gaplot1drange
%   as the plot function
%     options = gaoptimset('PlotFcns',@gaplot1drange);

%   Copyright 2012-2014 The MathWorks, Inc.
```

```
if isinf(options.Generations) || size(state.Population,2) > 1
    title('Plot Not Available','interp','none');
    return;
end
generation = state.Generation;
score = state.Population;
smean = mean(score);
Y = smean;
L = smean - min(score);
U = max(score) - smean;

switch flag

    case 'init'
        set(gca,'xlim',[1,options.Generations+1]);
        plotRange = errorbar(generation,Y,L,U);
        set(plotRange,'Tag','gaplot1drange');
        title('Range of Population, Mean','interp','none')
        xlabel('Generation','interp','none')
    case 'iter'
        plotRange = findobj(get(gca,'Children'),'Tag','gaplot1drange');
        newX = [get(plotRange,'Xdata') generation];
        newY = [get(plotRange,'Ydata') Y];
        newL = [get(plotRange,'Ldata'); L];
        newU = [get(plotRange,'Udata'); U];
        set(plotRange, 'Xdata',newX,'Ydata',newY,'Ldata',newL,'Udata',newU);
end
```

Note that all individuals lie between –70 and 70. The population never explores points near the global minimum at $x = 101$.

One way to make the genetic algorithm explore a wider range of points—that is, to increase the diversity of the populations—is to increase the **Initial range**. The **Initial range** does not have to include the point $x = 101$, but it must be large enough so that the algorithm generates individuals near $x = 101$. Set **Initial range** to `[-10;90]` as shown in the following figure.



Population

Population type: Double vector

Population size: [✓ Use default: 50 when numberOfVariables <= 5, else 200] Specify:

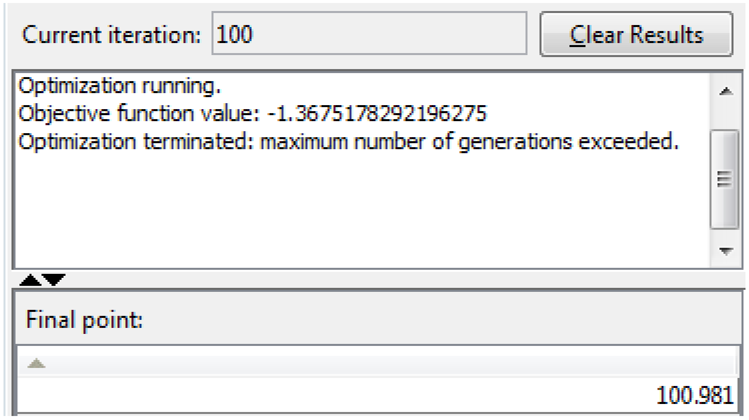
Creation function: Constraint dependent

Initial population: [✓ Use default: []] Specify:

Initial scores: [✓ Use default: []] Specify:

Initial range: Use default: [-10;10] [✓ Specify: [-10;90]]

Then click **Start**. The genetic algorithm returns a point very close to 101.



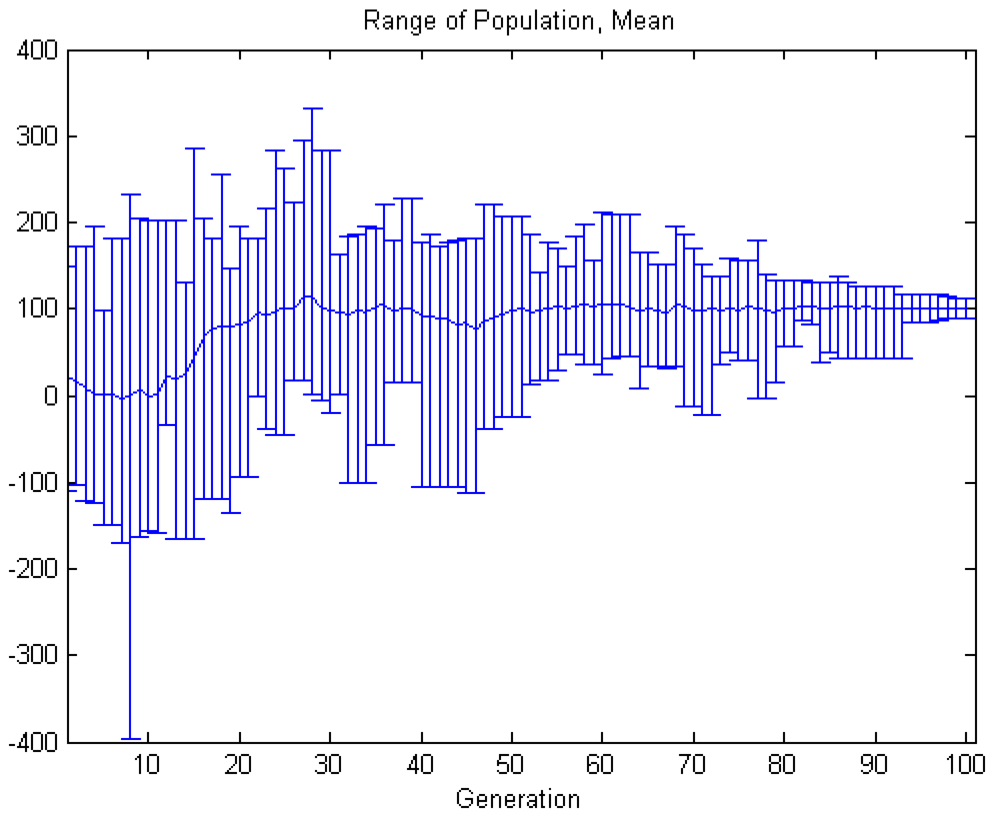
Current iteration: 100

```
Optimization running.
Objective function value: -1.3675178292196275
Optimization terminated: maximum number of generations exceeded.
```

Final point:

100.981

This time, the custom plot shows a much wider range of individuals. There are individuals near 101 from early on, and the population mean begins to converge to 101.


Range of Population, Mean
400
300
200
100
0
-100
-200
-300
-400
10
20
30
40
50
60
70
80
90
100
Generation

Include a Hybrid Function

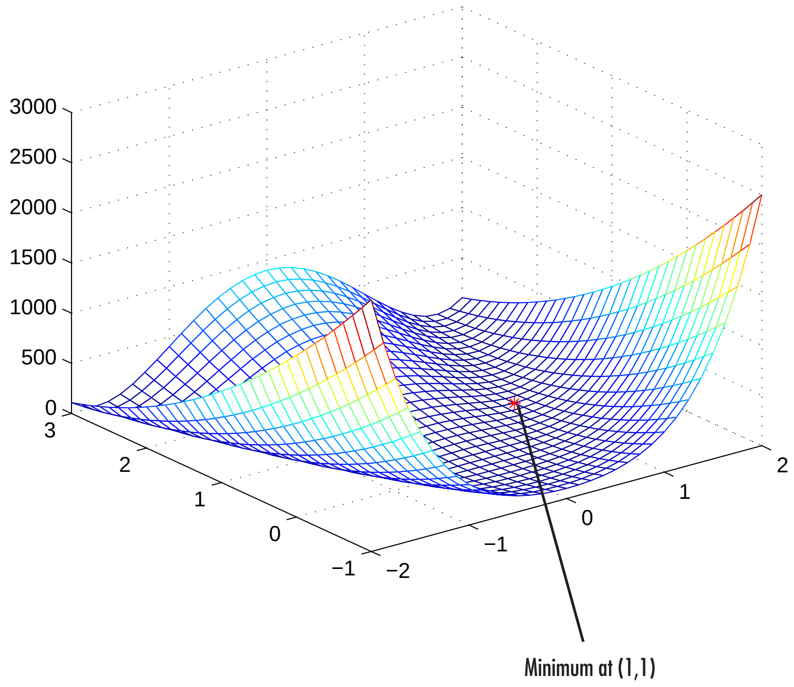
A hybrid function is an optimization function that runs after the genetic algorithm terminates in order to improve the value of the fitness function. The hybrid function uses the final point from the genetic algorithm as its initial point. You can specify a hybrid function in **Hybrid function** options.

This example uses Optimization Toolbox function `fminunc`, an unconstrained minimization function. The example first runs the genetic algorithm to find a point close to the optimal point and then uses that point as the initial point for `fminunc`.

The example finds the minimum of Rosenbrock's function, which is defined by

$$f(x_1,x_2) = 100\left(x_2 - x_1^2\right)^2 + (1-x_1)^2.$$

The following figure shows a plot of Rosenbrock's function.



Global Optimization Toolbox software contains the `dejong2fcn.m` file, which computes Rosenbrock's function. To see a worked example of a hybrid function, enter

```
hybriddemo
```

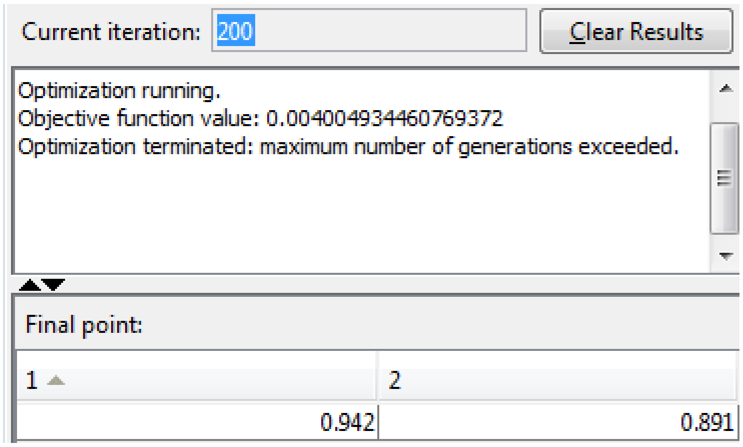
at the MATLAB prompt.

To explore the example, first enter `optimtool('ga')` to open the Optimization app to the `ga` solver. Enter the following settings:

- Set **Fitness function** to `@dejong2fcn`.
- Set **Number of variables** to `2`.
- Optionally, to get the same pseudorandom numbers as this example, switch to the command line and enter:

  ```
  rng(1,'twister')
  ```

Before adding a hybrid function, click **Start** to run the genetic algorithm by itself. The genetic algorithm displays the following results in the **Run solver and view results** pane:



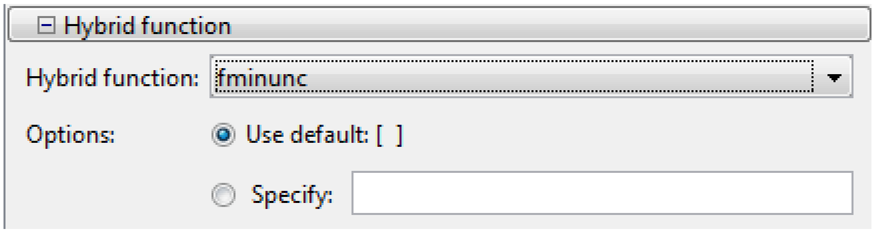
Current iteration: 200

```
Optimization running.
Objective function value: 0.004004934460769372
Optimization terminated: maximum number of generations exceeded.
```

Final point:

1	2
0.942	0.891

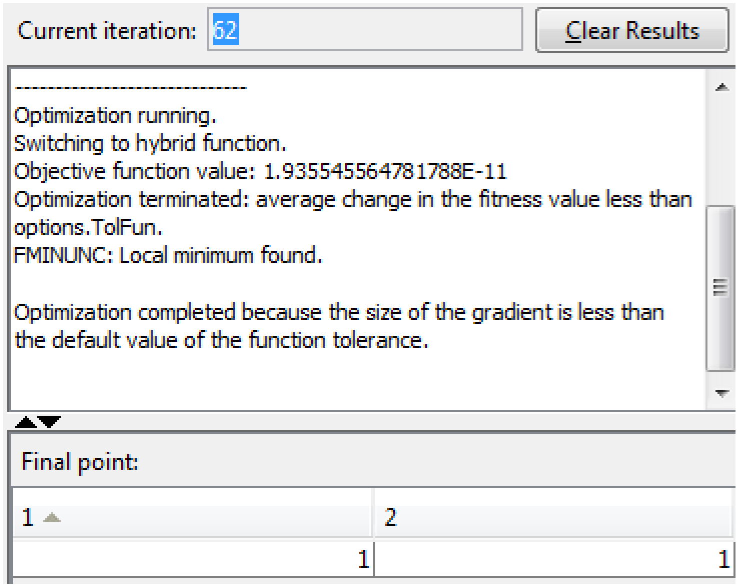
The final point is somewhat close to the true minimum at (1, 1). You can improve this result by setting **Hybrid function** to `fminunc` (in the **Hybrid function** options).



Hybrid function: fminunc

Options: [✓ Use default: [ ]] Specify:

`fminunc` uses the final point of the genetic algorithm as its initial point. It returns a more accurate result, as shown in the **Run solver and view results** pane.



Current iteration: 62

```
-----------------------------
Optimization running.
Switching to hybrid function.
Objective function value: 1.935545564781788E-11
Optimization terminated: average change in the fitness value less than
options.TolFun.
FMINUNC: Local minimum found.

Optimization completed because the size of the gradient is less than
the default value of the function tolerance.
```

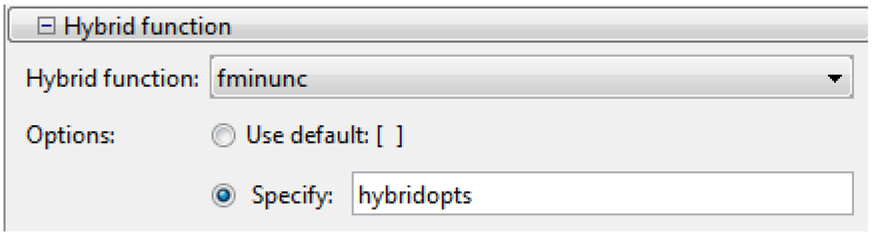
Final point:

1	2
1	1

Specify nondefault options for the hybrid function by creating options at the command line. Use `optimset` for `fminsearch`, `psoptimset` for `patternsearch`, or `optimoptions` for `fmincon` or `fminunc`. For example:

```
hybridopts = optimoptions('fminunc','Display','iter','Algorithm','quasi-newton');
```

In the Optimization app enter the name of your options structure in the **Options** box under **Hybrid function**:



Hybrid function

Hybrid function: fminunc

Options: Use default: [ ] [✓ Specify: hybridopts]

At the command line, the syntax is as follows:

```
options = gaoptimset('HybridFcn',{@fminunc,hybridopts});
```

`hybridopts` must exist before you set `options`.

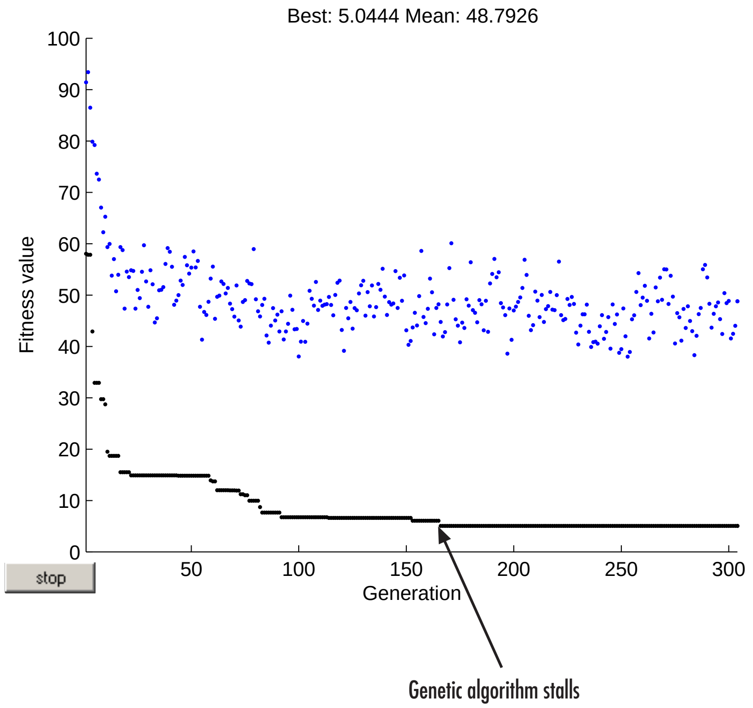
Set Maximum Number of Generations

The **Generations** option in **Stopping criteria** determines the maximum number of generations the genetic algorithm runs for—see “Stopping Conditions for the Algorithm” on page 5-22. Increasing the **Generations** option often improves the final result.

As an example, change the settings in the Optimization app as follows:

- Set **Fitness function** to `@rastriginsfcn`.
- Set **Number of variables** to `10`.
- Select **Best fitness** in the **Plot functions** pane.
- Set **Generations** to `Inf`.
- Set **Stall generations** to `Inf`.
- Set **Stall time limit** to `Inf`.

Run the genetic algorithm for approximately 300 generations and click **Stop**. The following figure shows the resulting best fitness plot after 300 generations.



Note that the algorithm *stalls* at approximately generation number 170—that is, there is no immediate improvement in the fitness function after generation 170. If you restore **Stall generations** to its default value of `50`, the algorithm could terminate at approximately generation number 220. If the genetic algorithm stalls repeatedly with the current setting for **Generations**, you can try increasing both the **Generations** and **Stall generations** options to improve your results. However, changing other options might be more effective.

Note When **Mutation function** is set to `Gaussian`, increasing the value of **Generations** might actually worsen the final result. This can occur because the Gaussian mutation function decreases the average amount of mutation in each generation by a factor that depends on the value specified in **Generations**. Consequently, the setting for **Generations** affects the behavior of the algorithm.

Vectorize the Fitness Function

In this section...
"Vectorize for Speed" on page 5-110
"Vectorized Constraints" on page 5-111

Vectorize for Speed

The genetic algorithm usually runs faster if you *vectorize* the fitness function. This means that the genetic algorithm only calls the fitness function once, but expects the fitness function to compute the fitness for all individuals in the current population at once. To vectorize the fitness function,

- Write the file that computes the function so that it accepts a matrix with arbitrarily many rows, corresponding to the individuals in the population. For example, to vectorize the function

  $$f(x_1, x_2) = x_1^2 - 2x_1x_2 + 6x_1 + x_2^2 - 6x_2$$

  write the file using the following code:

  ```
  z =x(:,1).^2 - 2*x(:,1).*x(:,2) + 6*x(:,1) + x(:,2).^2 - 6*x(:,2);
  ```

  The colon in the first entry of `x` indicates all the rows of `x`, so that `x(:, 1)` is a vector. The `.^` and `.*` operators perform elementwise operations on the vectors.
- In the **User function evaluation** pane, set the **Evaluate fitness and constraint functions** option to `vectorized`.

Note The fitness function, and any nonlinear constraint function, must accept an arbitrary number of rows to use the **Vectorize** option. `ga` sometimes evaluates a single row even during a vectorized calculation.

The following comparison, run at the command line, shows the improvement in speed with **Vectorize** set to `On`.

```
options = gaoptimset('PopulationSize',2000);
tic;ga(@rastriginsfcn,20,[],[],[],[],[],[],[],options);toc
```

```
Optimization terminated: maximum number of generations exceeded.
Elapsed time is 12.054973 seconds.

options=gaoptimset(options,'Vectorize','on');
tic;ga(@rastriginsfcn,20,[],[],[],[],[],[],[],options);toc
Optimization terminated: maximum number of generations exceeded.
Elapsed time is 1.860655 seconds.
```

Vectorized Constraints

If there are nonlinear constraints, the objective function and the nonlinear constraints all need to be vectorized in order for the algorithm to compute in a vectorized manner.

“Vectorize the Objective and Constraint Functions” on page 4-80 contains an example of how to vectorize both for the solver `patternsearch`. The syntax is nearly identical for `ga`. The only difference is that `patternsearch` can have its patterns appear as either row or column vectors; the corresponding vectors for `ga` are the population vectors, which are always rows.

Constrained Minimization Using ga

Suppose you want to minimize the simple fitness function of two variables x_1 and x_2,

$$\min_x f(x) = 100\left(x_1^2 - x_2\right)^2 + (1 - x_1)^2$$

subject to the following nonlinear inequality constraints and bounds

$$\begin{array}{ll} x_1 \cdot x_2 + x_1 - x_2 + 1.5 \le 0 & \text{(nonlinear constraint)} \\ 10 - x_1 \cdot x_2 \le 0 & \text{(nonlinear constraint)} \\ 0 \le x_1 \le 1 & \text{(bound)} \\ 0 \le x_2 \le 13 & \text{(bound)} \end{array}$$

Begin by creating the fitness and constraint functions. First, create a file named `simple_fitness.m` as follows:

```
function y = simple_fitness(x)
y = 100*(x(1)^2 - x(2))^2 + (1 - x(1))^2;
```

(`simple_fitness.m` ships with Global Optimization Toolbox software.)

The genetic algorithm function, `ga`, assumes the fitness function will take one input `x`, where `x` has as many elements as the number of variables in the problem. The fitness function computes the value of the function and returns that scalar value in its one return argument, `y`.

Then create a file, `simple_constraint.m`, containing the constraints

```
function [c, ceq] = simple_constraint(x)
c = [1.5 + x(1)*x(2) + x(1) - x(2);...
-x(1)*x(2) + 10];
ceq = [];
```

(`simple_constraint.m` ships with Global Optimization Toolbox software.)

The `ga` function assumes the constraint function will take one input `x`, where `x` has as many elements as the number of variables in the problem. The constraint function computes the values of all the inequality and equality constraints and returns two vectors, `c` and `ceq`, respectively.

To minimize the fitness function, you need to pass a function handle to the fitness function as the first argument to the ga function, as well as specifying the number of variables as the second argument. Lower and upper bounds are provided as LB and UB respectively. In addition, you also need to pass a function handle to the nonlinear constraint function.

```
ObjectiveFunction = @simple_fitness;
nvars = 2;    % Number of variables
LB = [0 0];   % Lower bound
UB = [1 13];  % Upper bound
ConstraintFunction = @simple_constraint;
rng(1,'twister') % for reproducibility
[x,fval] = ga(ObjectiveFunction,nvars,...
    [],[],[],[],LB,UB,ConstraintFunction)

Optimization terminated: average change in the fitness value
less than options.TolFun and constraint violation is
less than options.TolCon.

x =

    0.8123   12.3104


fval =

   1.3574e+04
```

The genetic algorithm solver handles linear constraints and bounds differently from nonlinear constraints. All the linear constraints and bounds are satisfied throughout the optimization. However, ga may not satisfy all the nonlinear constraints at every generation. If ga converges to a solution, the nonlinear constraints will be satisfied at that solution.

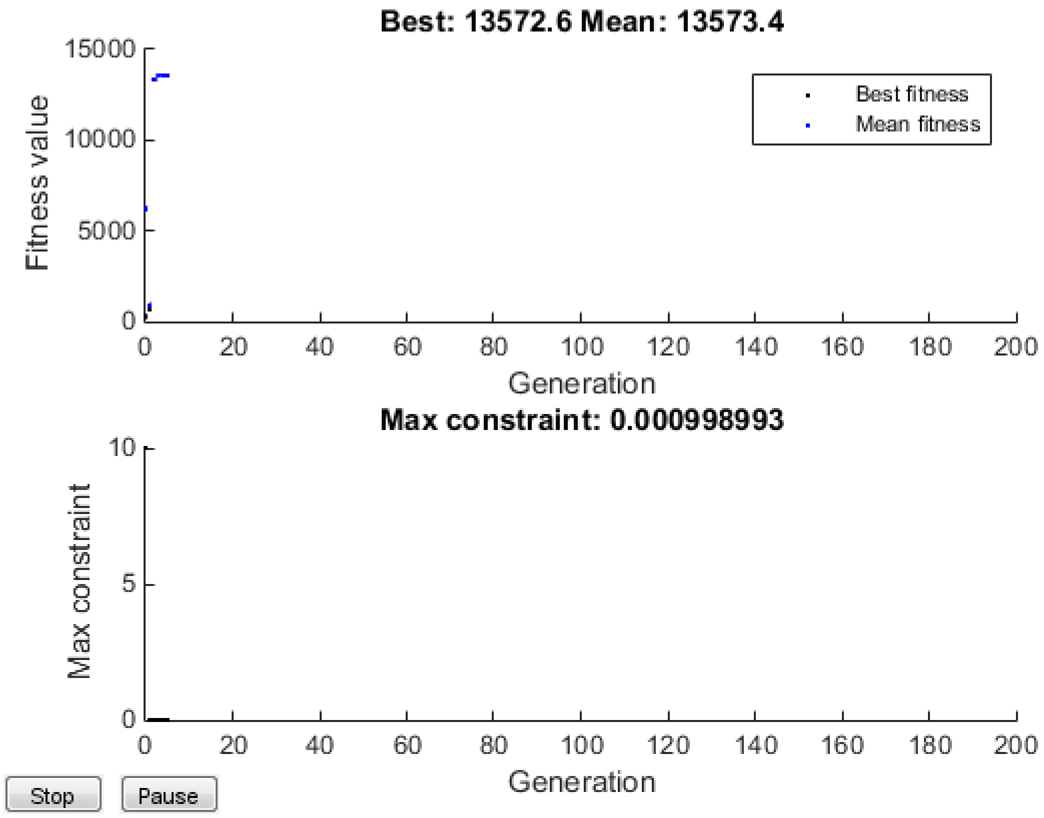
ga uses the mutation and crossover functions to produce new individuals at every generation. ga satisfies linear and bound constraints by using mutation and crossover functions that only generate feasible points. For example, in the previous call to ga, the mutation function mutationguassian does not necessarily obey the bound constraints. So when there are bound or linear constraints, the default ga mutation function is mutationadaptfeasible. If you provide a custom mutation function, this custom function must only generate points that are feasible with respect to the linear and bound constraints. All the included crossover functions generate points that satisfy the linear constraints and bounds except the crossoverheuristic function.

To see the progress of the optimization, use the `gaoptimset` function to create an options structure that selects two plot functions. The first plot function is `gaplotbestf`, which plots the best and mean score of the population at every generation. The second plot function is `gaplotmaxconstr`, which plots the maximum constraint violation of nonlinear constraints at every generation. You can also visualize the progress of the algorithm by displaying information to the command window using the `'Display'` option.

```
options = gaoptimset('PlotFcns',{@gaplotbestf,@gaplotmaxconstr},'Display','iter');
```

Rerun the `ga` solver.

```
[x,fval] = ga(ObjectiveFunction,nvars,[],[],[],[],...
              LB,UB,ConstraintFunction,options)
                                 Best       max        Stall
    Generation  f-count         f(x)     constraint  Generations
        1        2670       13666.1           0      0
        2        5282       13578.2    6.648e-05      0
        3        7894       14043.7           0      0
        4       13156         14045           0      0
        5       17806       13573.5     0.000999      0
    Optimization terminated: average change in the fitness value less than options.TolFun
     and constraint violation is less than options.TolCon.

    x =

        0.8122   12.3103


    fval =

       1.3573e+04
```



You can provide a start point for the minimization to the `ga` function by specifying the `InitialPopulation` option. The `ga` function will use the first individual from `InitialPopulation` as a start point for a constrained minimization.

```
X0 = [0.5 0.5]; % Start point (row vector)
options = gaoptimset(options,'InitialPopulation',X0);
```

Now, rerun the `ga` solver.

```
[x,fval] = ga(ObjectiveFunction,nvars,[],[],[],[],...
              LB,UB,ConstraintFunction,options)
                            Best        max        Stall
Generation  f-count         f(x)     constraint  Generations
    1        2670       13578.1    0.0005253       0
    2        5282       13578.2    1.479e-05       0
    3        8394       14037.2            0       0
    4       17094       13573.6    0.0009898      0
Optimization terminated: average change in the fitness value less than options.TolFun
 and constraint violation is less than options.TolCon.

x =

    0.8123   12.3103


fval =

   1.3574e+04
```

Particle Swarm Optimization

What Is Particle Swarm Optimization?

Particle swarm is a population-based algorithm. In this respect it is similar to the genetic algorithm. A collection of individuals called particles move in steps throughout a region. At each step, the algorithm evaluates the objective function at each particle. After this evaluation, the algorithm decides on the new velocity of each particle. The particles move, then the algorithm reevaluates.

The inspiration for the algorithm is flocks of birds or insects swarming. Each particle is attracted to some degree to the best location it has found so far, and also to the best location any member of the swarm has found. After some steps, the population can coalesce around one location, or can coalesce around a few locations, or can continue to move.

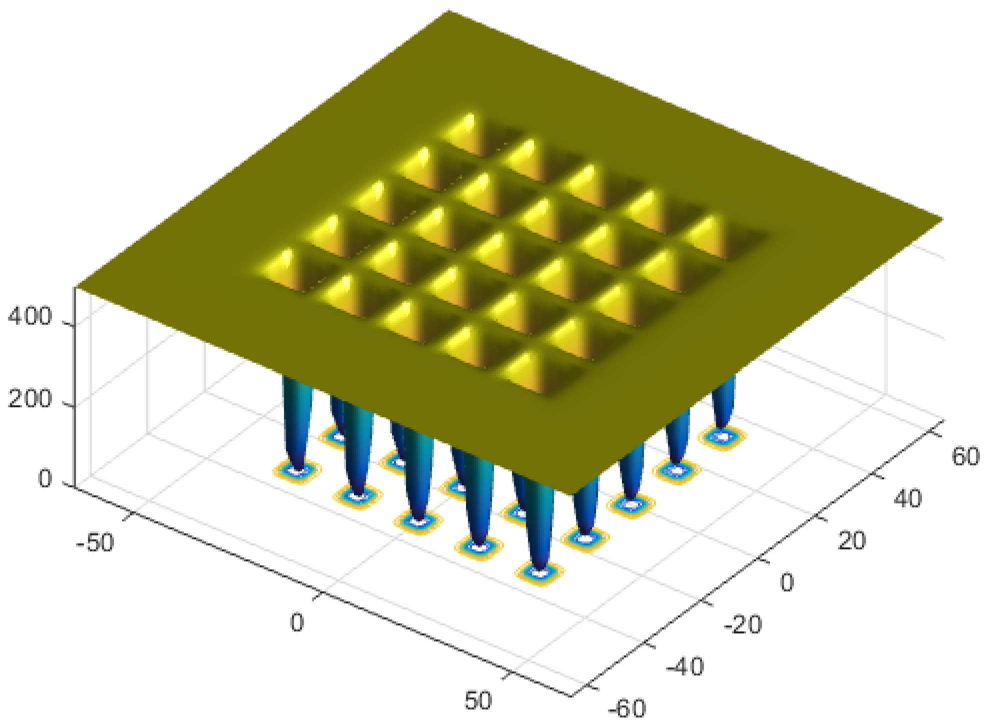
The `particleswarm` function attempts to optimize using a "Particle Swarm Optimization Algorithm" on page 6-10.

Optimize Using Particle Swarm

This example shows how to optimize using the `particleswarm` solver.

The objective function in this example is De Jong’s fifth function, which is included with Global Optimization Toolbox software.

```
dejong5fcn
```



This function has 25 local minima.

Try to find the minimum of the function using the default `particleswarm` settings.

```
fun = @dejong5fcn;
nvars = 2;
rng default % For reproducibility
[x,fval,exitflag] = particleswarm(fun,nvars)

Optimization ended: relative change in the objective value
over the last OPTIONS.StallIterLimit iterations is less than OPTIONS.TolFun.

x =

  -31.9521  -16.0176


fval =

    5.9288


exitflag =

     1
```

Is the solution `x` the global optimum? It is unclear at this point. Looking at the function plot shows that the function has local minima for components in the range `[-50,50]`. So restricting the range of the variables to `[-50,50]` helps the solver locate a global minimum.

```
lb = [-50;-50];
ub = -lb;
[x,fval,exitflag] = particleswarm(fun,nvars,lb,ub)

x =

  -16.0079  -31.9697


fval =

    1.9920


exitflag =

     1
```

This looks promising: the new solution has lower `fval` than the previous one. But is `x` truly a global solution? Try minimizing again with more particles, to better search the region.

```
options = optimoptions('particleswarm','SwarmSize',100);
[x,fval,exitflag] = particleswarm(fun,nvars,lb,ub,options)
```

```
Optimization ended: change in the objective value less than options.TolFun.

x =

  -31.9781  -31.9784


fval =

    0.9980


exitflag =

     1
```

This looks even more promising. But is this answer a global solution, and how accurate is it? Rerun the solver with a hybrid function. `particleswarm` calls the hybrid function after `particleswarm` finishes its iterations.

```
options.HybridFcn = @fmincon;
[x,fval,exitflag] = particleswarm(fun,nvars,lb,ub,options)
```

```
Optimization ended: change in the objective value less than options.TolFun.

x =

  -31.9783  -31.9784


fval =

    0.9980


exitflag =

     1
```

`particleswarm` found essentially the same solution as before. This gives you some confidence that `particleswarm` reports a local minimum and that the final `x` is the global solution.

Particle Swarm Output Function

This example shows how to use an output function for `particleswarm`. The output function plots the range that the particles occupy in each dimension.

An output function runs after each iteration of the solver. For syntax details, and for the data available to an output function, see the `particleswarm` options reference pages.

Copy the following code into a file named `pswplotranges.m` on your MATLAB path. The code sets up `nplot` subplots, where `nplot` is the number of dimensions in the problem.

```
function stop = pswplotranges(optimValues,state)

stop = false; % This function does not stop the solver
switch state
    case 'init'
        nplot = size(optimValues.swarm,2); % Number of dimensions
        for i = 1:nplot % Set up axes for plot
            subplot(nplot,1,i);
            tag = sprintf('psoplotrange_var_%g',i); % Set a tag for the subplot
            semilogy(optimValues.iteration,0,'-k','Tag',tag); % Log-scaled plot
            ylabel(num2str(i))
        end
        xlabel('Iteration','interp','none'); % Iteration number at the bottom
        subplot(nplot,1,1) % Title at the top
        title('Log range of particles by component')
        setappdata(gcf,'t0',tic); % Set up a timer to plot only when needed
    case 'iter'
        nplot = size(optimValues.swarm,2); % Number of dimensions
        for i = 1:nplot
            subplot(nplot,1,i);
            % Calculate the range of the particles at dimension i
            irange = max(optimValues.swarm(:,i)) - min(optimValues.swarm(:,i));
            tag = sprintf('psoplotrange_var_%g',i);
            plotHandle = findobj(get(gca,'Children'),'Tag',tag); % Get the subplot
            xdata = plotHandle.XData; % Get the X data from the plot
            newX = [xdata optimValues.iteration]; % Add the new iteration
            plotHandle.XData = newX; % Put the X data into the plot
            ydata = plotHandle.YData; % Get the Y data from the plot
            newY = [ydata irange]; % Add the new value
            plotHandle.YData = newY; % Put the Y data into the plot
        end
        if toc(getappdata(gcf,'t0')) > 1/30 % If 1/30 s has passed
          drawnow % Show the plot
          setappdata(gcf,'t0',tic); % Reset the timer
        end
    case 'done'
        % No cleanup necessary
end
```

This output function draws a plot with one line per dimension. Each line represents the range of the particles in the swarm in that dimension. The plot is log-scaled to accommodate wide ranges. If the swarm converges to a single point, then the range of each dimension goes to zero. But if the swarm does not converge to a single point, then the range stays away from zero in some dimensions.

Define the problem in your workspace. The `multirosenbrock` function is a generalization of Rosenbrock's function to any even number of dimensions. It has a global minimum of `0` at the point `[1,1,1,1,...]`. The `multirosenbrock.m` file is included with Global Optimization Toolbox software. See its code by entering `type multirosenbrock`.

```
fun = @multirosenbrock;
nvar = 4; % A 4-D problem
lb = -10*ones(nvar,1); % Bounds to help the solver converge
ub = -lb;
```

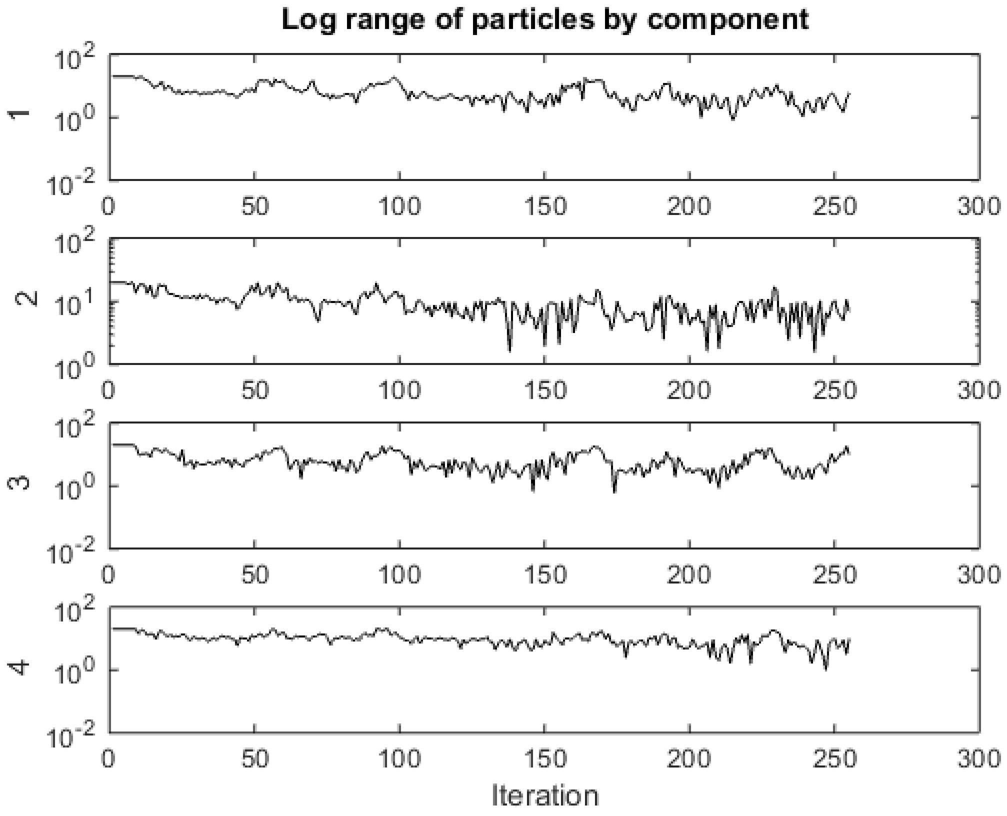
Set the options to use the output function.

```
options = optimoptions(@particleswarm,'OutputFcns',@pswplotranges);
```

Set the random number generator to get reproducible output. Then call the solver.

```
rng default % For reproducibility
[x,fval,eflag] = particleswarm(fun,nvar,lb,ub,options)
```

```
Optimization ended: relative change in the objective value
over the last OPTIONS.StallIterLimit iterations is less than OPTIONS.TolFun.

x =

    0.9964    0.9930    0.9835    0.9681


fval =
     3.4935e-04


eflag =

     1
```



The solver returned a point near the optimum `[1,1,1,1]`. But the span of the swarm did not converge to zero.

More About

- "Output Function and Plot Function" on page 10-57

Particle Swarm Optimization Algorithm

In this section...

Algorithm Outline

The particle swarm algorithm begins by creating the initial particles, and assigning them initial velocities.

It evaluates the objective function at each particle location, and determines the best (lowest) function value and the best location.

It chooses new velocities, based on the current velocity, the particles’ individual best locations, and the best locations of their neighbors.

It then iteratively updates the particle locations (the new location is the old one plus the velocity, modified to keep particles within bounds), velocities, and neighbors.

Iterations proceed until the algorithm reaches a stopping criterion.

Here are the details of the steps.

Initialization

By default, `particleswarm` creates particles at random uniformly within bounds. If there is an unbounded component, `particleswarm` creates particles with a random uniform distribution from –1000 to 1000. If you have only one bound, `particleswarm` shifts the creation to have the bound as an endpoint, and a creation interval 2000 wide. Particle `i` has position `x(i)`, which is a row vector with `nvars` elements. Control the span of the initial swarm using the `InitialSwarmSpan` option.

Similarly, `particleswarm` creates initial particle velocities `v` uniformly within the range `[-r,r]`, where `r` is the vector of initial *ranges*. The range of component `i` is the `ub(i) - lb(i)`, but for unbounded or semi-unbounded components the range is the `InitialSwarmSpan` option.

`particleswarm` evaluates the objective function at all particles. It records the current position `p(i)` of each particle `i`. In subsequent iterations, `p(i)` will be the location of the best objective function that particle `i` has found. And `b` is the best over all particles: `b = min(fun(p(i)))`. `d` is the location such that `b = fun(d)`.

`particleswarm` initializes the neighborhood size `N` to `minNeighborhoodSize = max(1,floor(SwarmSize*minFractionNeighbors))`.

`particleswarm` initializes the inertia `W = max(InertiaRange)`, or if `InertiaRange` is negative, it sets `W = min(InertiaRange)`.

`particleswarm` initializes the stall counter `c = 0`.

For convenience of notation, set the variable `y1 = SelfAdjustment`, and `y2 = SocialAdjustment`, where `SelfAdjustment` and `SocialAdjustment` are options.

Iteration Steps

The algorithm updates the swarm as follows. For particle `i`, which is at position `x(i)`:

1. Choose a random subset `S` of `N` particles other than `i`.
2. Find `fbest(S)`, the best objective function among the neighbors, and `g(S)`, the position of the neighbor with the best objective function.
3. For `u1` and `u2` uniformly (0,1) distributed random vectors of length `nvars`, update the velocity
   `v = W*v + y1*u1.*(p-x) + y2*u2.*(g-x)`.

   This update uses a weighted sum of:

   - The previous velocity `v`
   - The difference between the current position and the best position the particle has seen `p-x`
   - The difference between the current position and the best position in the current neighborhood `g-x`
4. Update the position `x = x + v`.
5. Enforce the bounds. If any component of `x` is outside a bound, set it equal to that bound.
6. Evaluate the objective function `f = fun(x)`.

7 If `f < fun(p)`, then set `p = x`. This step ensures `p` has the best position the particle has seen.

8 If `f < b`, then set `b = f` and `d = x`. This step ensures `b` has the best objective function in the swarm, and `d` has the best location.

9 If, in the previous step, the best function value was lowered, then set `flag = true`. Otherwise, `flag = false`. The value of `flag` is used in the next step.

10 Update the neighborhood. If `flag = true`:

  a Set `c = max(0,c-1)`.

  b Set `N` to `minNeighborhoodSize`.

  c If `c < 2`, then set `W = 2*W`.

  d If `c > 5`, then set `W = W/2`.

  e Ensure that `W` is in the bounds of the `InertiaRange` option.

  If `flag = false`:

  a Set `c = c+1`.

  b Set `N = min(N + minNeighborhoodSize,SwarmSize)`.

Stopping Criteria

`particleswarm` iterates until it reaches a stopping criterion.

Stopping Option	Stopping Test	Exit Flag
`StallIterLimit` and `TolFun`	Relative change in the best objective function value `g` over the last `StallIterLimit` iterations is less than `TolFun`.	1
`MaxIter`	Number of iterations reaches `MaxIter`.	0
`OutputFcns` or `PlotFcns`	`OutputFcns` or `PlotFcns` can halt the iterations.	-1
`ObjectiveLimit`	Best objective function value `g` is less than or equal to `ObjectiveLimit`.	-3

Stopping Option	Stopping Test	Exit Flag
`StallTimeLimit`	Best objective function value `g` did not change in the last `StallTimeLimit` seconds.	`-4`
`MaxTime`	Function run time exceeds `MaxTime` seconds.	`-5`

If `particleswarm` stops with exit flag `1`, it optionally calls a hybrid function after it exits.

Using Simulated Annealing

What Is Simulated Annealing?

Simulated annealing is a method for solving unconstrained and bound-constrained optimization problems. The method models the physical process of heating a material and then slowly lowering the temperature to decrease defects, thus minimizing the system energy.

At each iteration of the simulated annealing algorithm, a new point is randomly generated. The distance of the new point from the current point, or the extent of the search, is based on a probability distribution with a scale proportional to the temperature. The algorithm accepts all new points that lower the objective, but also, with a certain probability, points that raise the objective. By accepting points that raise the objective, the algorithm avoids being trapped in local minima, and is able to explore globally for more possible solutions. An *annealing schedule* is selected to systematically decrease the temperature as the algorithm proceeds. As the temperature decreases, the algorithm reduces the extent of its search to converge to a minimum.

Optimize Using Simulated Annealing

In this section...

Calling simulannealbnd at the Command Line

To call the simulated annealing function at the command line, use the syntax

```
[x fval] = simulannealbnd(@objfun,x0,lb,ub,options)
```

where

- `@objfun` is a function handle to the objective function.
- `x0` is an initial guess for the optimizer.
- `lb` and `ub` are lower and upper bound constraints, respectively, on `x`.
- `options` is a structure containing options for the algorithm. If you do not pass in this argument, `simulannealbnd` uses its default options.

The results are given by:

- `x` — Final point returned by the solver
- `fval` — Value of the objective function at `x`

The command-line function `simulannealbnd` is convenient if you want to

- Return results directly to the MATLAB workspace.
- Run the simulated annealing algorithm multiple times with different options by calling `simulannealbnd` from a file.

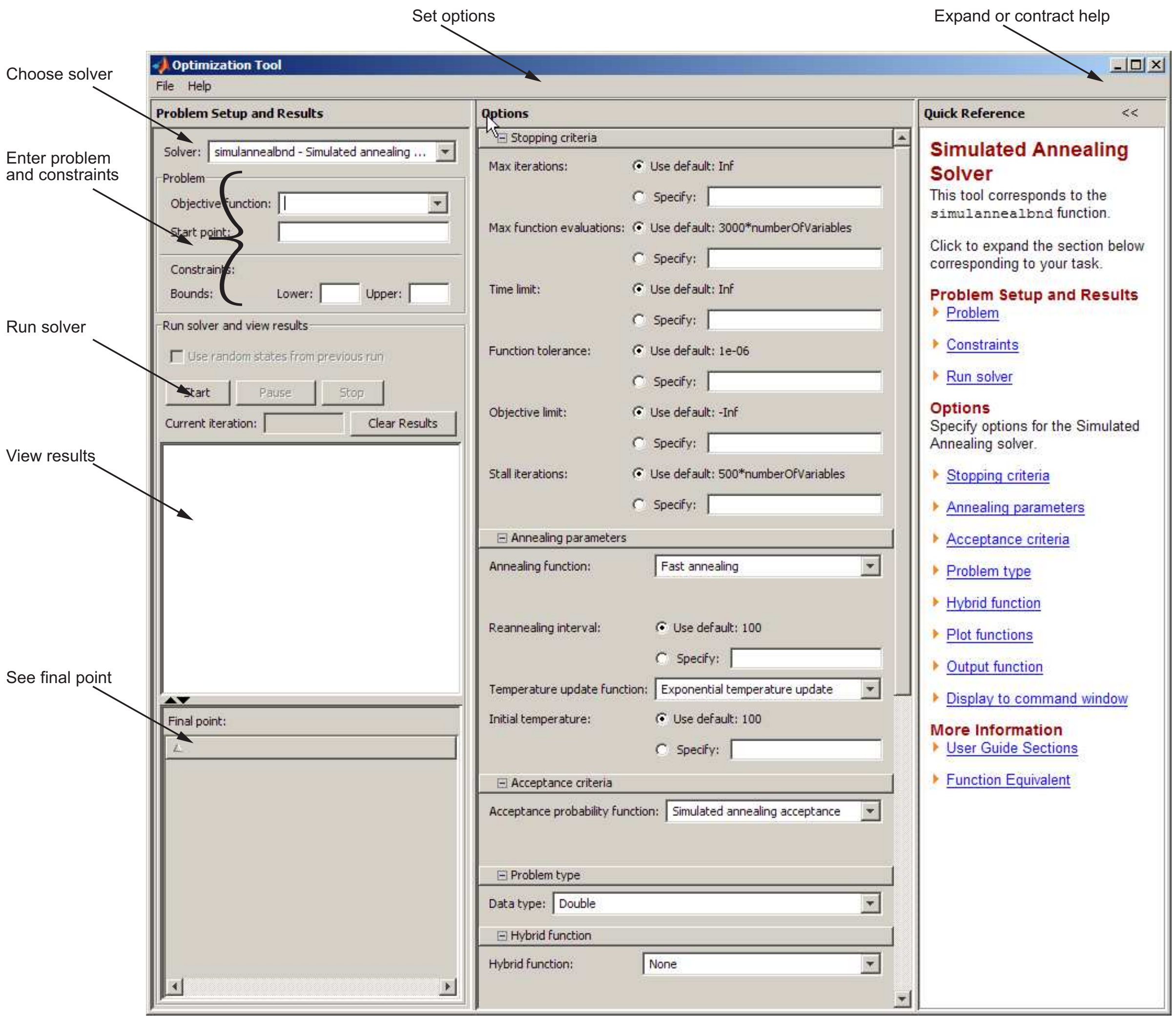
"Command Line Simulated Annealing Optimization" on page 7-15 provides a detailed description of using the function `simulannealbnd` and creating the options structure.

Using the Optimization App

To open the Optimization app, enter

```
optimtool('simulannealbnd')
```

at the command line, or enter `optimtool` and then choose `simulannealbnd` from the **Solver** menu.



Optimization Tool

File Help

Problem Setup and Results

Solver: simulannealbnd - Simulated annealing ...

Problem

Objective function:

Start point:

Constraints:

Bounds: Lower: Upper:

Run solver and view results

Use random states from previous run

Start Pause Stop

Current iteration: Clear Results

Final point:

Options

Stopping criteria

Max iterations: Use default: Inf

Max function evaluations: Use default: 3000*numberOfVariables

Time limit: Use default: Inf

Function tolerance: Use default: 1e-06

Objective limit: Use default: -Inf

Stall iterations: Use default: 500*numberOfVariables

Annealing parameters

Annealing function: Fast annealing

Reannealing interval: Use default: 100

Temperature update function: Exponential temperature update

Initial temperature: Use default: 100

Acceptance criteria

Acceptance probability function: Simulated annealing acceptance

Problem type

Data type: Double

Hybrid function

Hybrid function: None

Quick Reference

Simulated Annealing Solver

This tool corresponds to the `simulannealbnd` function.

Click to expand the section below corresponding to your task.

Problem Setup and Results

- Problem
- Constraints
- Run solver

Options

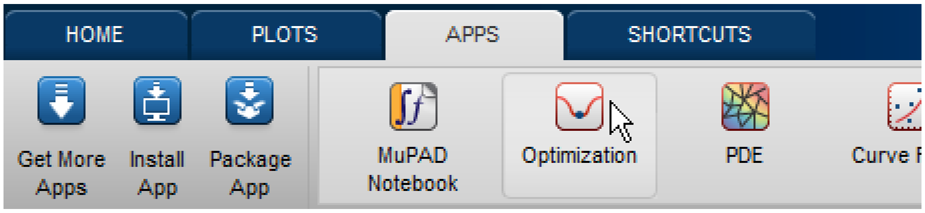
Specify options for the Simulated Annealing solver.

- Stopping criteria
- Annealing parameters
- Acceptance criteria
- Problem type
- Hybrid function
- Plot functions
- Output function
- Display to command window

More Information

- User Guide Sections
- Function Equivalent

You can also start the tool from the MATLAB **Apps** tab.



To use the Optimization app, you must first enter the following information:

- **Objective function** — The objective function you want to minimize. Enter the fitness function in the form `@fitnessfun`, where `fitnessfun.m` is a file that computes the objective function. "Compute Objective Functions" on page 2-2 explains how write this file. The `@` sign creates a function handle to `fitnessfun`.
- **Number of variables** — The length of the input vector to the fitness function. For the function `my_fun` described in "Compute Objective Functions" on page 2-2, you would enter `2`.

You can enter bounds for the problem in the **Constraints** pane. If the problem is unconstrained, leave these fields blank.

To run the simulated annealing algorithm, click the **Start** button. The tool displays the results of the optimization in the **Run solver and view results** pane.

You can change the options for the simulated annealing algorithm in the **Options** pane. To view the options in one of the categories listed in the pane, click the + sign next to it.

For more information,

- See "Optimization App" in the Optimization Toolbox documentation.
- See "Minimize Using the Optimization App" on page 7-8 for an example of using the tool with the function `simulannealbnd`.

Minimize Function with Many Local Minima

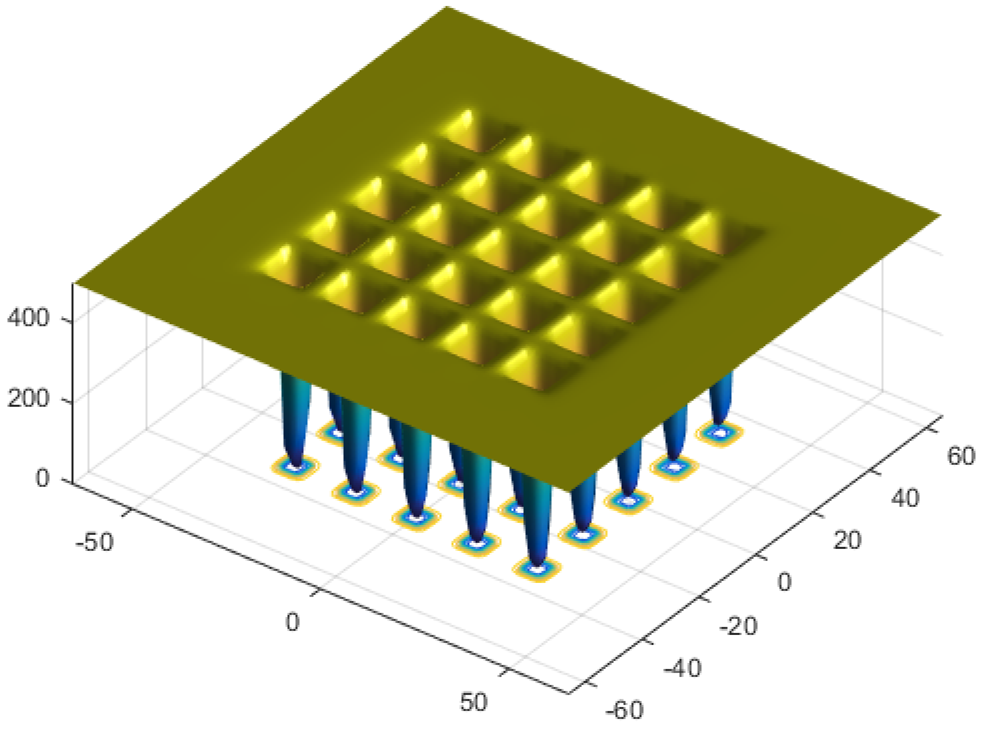
In this section...

Description

This section presents an example that shows how to find a local minimum of a function using simulated annealing.

De Jong's fifth function is a two-dimensional function with many (25) local minima:

```
dejong5fcn
```



Many standard optimization algorithms get stuck in local minima. Because the simulated annealing algorithm performs a wide random search, the chance of being trapped in local minima is decreased.

Note: Because simulated annealing uses random number generators, each time you run this algorithm you can get different results. See "Reproduce Your Results" on page 7-18 for more information.

Minimize at the Command Line

To run the simulated annealing algorithm without constraints, call `simulannealbnd` at the command line using the objective function in `dejong5fcn.m`, referenced by anonymous function pointer:

```
rng(10,'twister') % for reproducibility
fun = @dejong5fcn;
[x fval] = simulannealbnd(fun,[0 0])
```

This returns

```
Optimization terminated: change in best function value less than options.TolFun.

x =
  -16.1292  -15.8214

fval =
    6.9034
```

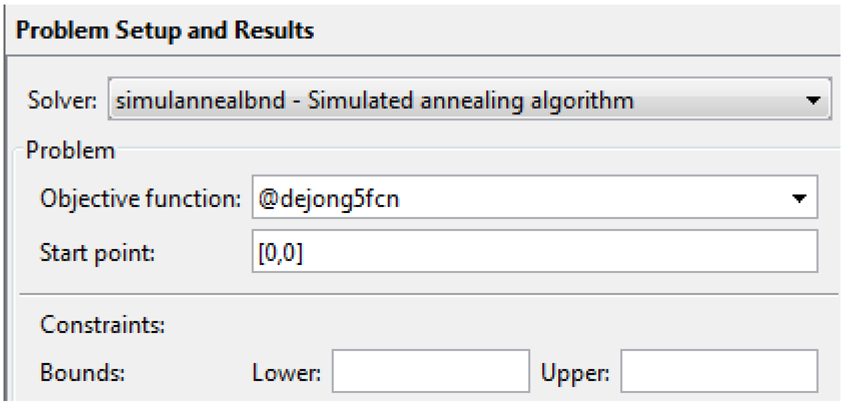
where

- `x` is the final point returned by the algorithm.
- `fval` is the objective function value at the final point.

Minimize Using the Optimization App

To run the minimization using the Optimization app,

1 Set up your problem as pictured in the Optimization app



Problem Setup and Results

Solver: simulannealbnd - Simulated annealing algorithm

Problem

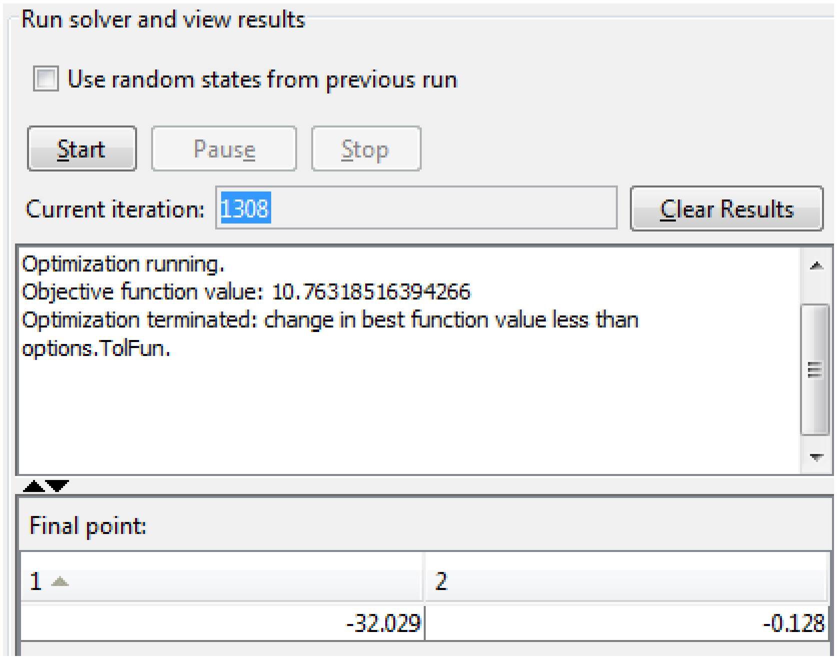
Objective function: @dejong5fcn

Start point: [0,0]

Constraints:

Bounds: Lower: Upper:

2 Click **Start** under **Run solver and view results**:



Run solver and view results

Use random states from previous run

Current iteration: 1308

```
Optimization running.
Objective function value: 10.76318516394266
Optimization terminated: change in best function value less than
options.TolFun.
```

Final point:

1	2
-32.029	-0.128

Your results can differ from the pictured ones, because `simulannealbnd` uses a random number stream.

Simulated Annealing Terminology

In this section...

Objective Function

The *objective function* is the function you want to optimize. Global Optimization Toolbox algorithms attempt to find the minimum of the objective function. Write the objective function as a file or anonymous function, and pass it to the solver as a `function_handle`. For more information, see "Compute Objective Functions" on page 2-2.

Temperature

The *temperature* is a parameter in simulated annealing that affects two aspects of the algorithm:

- The distance of a trial point from the current point (See "Outline of the Algorithm" on page 7-12, Step 1.)
- The probability of accepting a trial point with higher objective function value (See "Outline of the Algorithm" on page 7-12, Step 2.)

Temperature can be a vector with different values for each component of the current point. Typically, the initial temperature is a scalar.

Temperature decreases gradually as the algorithm proceeds. You can specify the initial temperature as a positive scalar or vector in the `InitialTemperature` option. You can specify the temperature as a function of iteration number as a function handle in the `TemperatureFcn` option. The temperature is a function of the "Annealing Parameter" on page 7-11, which is a proxy for the iteration number. The slower the rate of temperature decrease, the better the chances are of finding an optimal solution, but the longer the run time. For a list of built-in temperature functions and the syntax of a custom temperature function, see "Temperature Options" on page 10-64.

Annealing Parameter

The *annealing parameter* is a proxy for the iteration number. The algorithm can raise temperature by setting the annealing parameter to a lower value than the current iteration. (See “Reannealing” on page 7-11.) You can specify the temperature schedule as a function handle with the `TemperatureFcn` option.

Reannealing

Annealing is the technique of closely controlling the temperature when cooling a material to ensure that it reaches an optimal state. *Reannealing* raises the temperature after the algorithm accepts a certain number of new points, and starts the search again at the higher temperature. Reannealing avoids the algorithm getting caught at local minima. Specify the reannealing schedule with the `ReannealInterval` option.

How Simulated Annealing Works

In this section...
"Outline of the Algorithm" on page 7-12
"Stopping Conditions for the Algorithm" on page 7-14
"Bibliography" on page 7-14

Outline of the Algorithm

The simulated annealing algorithm performs the following steps:

1. The algorithm generates a random trial point. The algorithm chooses the distance of the trial point from the current point by a probability distribution with a scale depending on the current temperature. You set the trial point distance distribution as a function with the `AnnealingFcn` option. Choices:

   - `@annealingfast` (default) — Step length equals the current temperature, and direction is uniformly random.
   - `@annealingboltz` — Step length equals the square root of temperature, and direction is uniformly random.
   - `@myfun` — Custom annealing algorithm, `myfun`. For custom annealing function syntax, see "Algorithm Settings" on page 10-65.

2. The algorithm determines whether the new point is better or worse than the current point. If the new point is better than the current point, it becomes the next point. If the new point is worse than the current point, the algorithm can still make it the next point. The algorithm accepts a worse point based on an acceptance function. Choose the acceptance function with the `AcceptanceFcn` option. Choices:

   - `@acceptancesa` (default) — Simulated annealing acceptance function. The probability of acceptance is

$$\frac{1}{1+\exp\left(\dfrac{\Delta}{\max(T)}\right)},$$

   where

Δ = new objective – old objective.
T_0 = initial temperature of component i
T = the current temperature.

Since both Δ and T are positive, the probability of acceptance is between 0 and 1/2. Smaller temperature leads to smaller acceptance probability. Also, larger Δ leads to smaller acceptance probability.

- `@myfun` — Custom acceptance function, `myfun`. For custom acceptance function syntax, see “Algorithm Settings” on page 10-65.

3 The algorithm systematically lowers the temperature, storing the best point found so far. The `TemperatureFcn` option specifies the function the algorithm uses to update the temperature. Let k denote the annealing parameter. (The annealing parameter is the same as the iteration number until reannealing.) Options:

- `@temperatureexp` (default) — $T = T_0 * 0.95\hat{}k$.
- `@temperaturefast` — $T = T_0 / k$.
- `@temperatureboltz` — $T = T_0 / \log(k)$.
- `@myfun` — Custom temperature function, `myfun`. For custom temperature function syntax, see “Temperature Options” on page 10-64.

4 `simulannealbnd` reanneals after it accepts `ReannealInterval` points. Reannealing sets the annealing parameters to lower values than the iteration number, thus raising the temperature in each dimension. The annealing parameters depend on the values of estimated gradients of the objective function in each dimension. The basic formula is

$$k_i = \log\left(\frac{T_0}{T_i}\frac{\max\limits_j(s_j)}{s_i}\right),$$

where
k_i = annealing parameter for component i.
T_0 = initial temperature of component i.
T_i = current temperature of component i.
s_i = gradient of objective in direction i times difference of bounds in direction i.

`simulannealbnd` safeguards the annealing parameter values against `Inf` and other improper values.

5 The algorithm stops when the average change in the objective function is small relative to the `TolFun` tolerance, or when it reaches any other stopping criterion. See "Stopping Conditions for the Algorithm" on page 7-14.

For more information on the algorithm, see Ingber [1].

Stopping Conditions for the Algorithm

The simulated annealing algorithm uses the following conditions to determine when to stop:

- `TolFun` — The algorithm runs until the average change in value of the objective function in `StallIterLim` iterations is less than the value of `TolFun`. The default value is `1e-6`.
- `MaxIter` — The algorithm stops when the number of iterations exceeds this maximum number of iterations. You can specify the maximum number of iterations as a positive integer or `Inf`. The default value is `Inf`.
- `MaxFunEval` specifies the maximum number of evaluations of the objective function. The algorithm stops if the number of function evaluations exceeds the value of `MaxFunEval`. The default value is `3000*numberofvariables`.
- `TimeLimit` specifies the maximum time in seconds the algorithm runs before stopping. The default value is `Inf`.
- `ObjectiveLimit` — The algorithm stops when the best objective function value is less than or equal to the value of `ObjectiveLimit`. The default value is `-Inf`.

Bibliography

[1] Ingber, L. *Adaptive simulated annealing (ASA): Lessons learned*. Invited paper to a special issue of the *Polish Journal Control and Cybernetics* on "Simulated Annealing Applied to Combinatorial Optimization." 1995. Available from http://www.ingber.com/asa96_lessons.ps.gz

Command Line Simulated Annealing Optimization

In this section...

Run simulannealbnd With the Default Options

To run the simulated annealing algorithm with the default options, call `simulannealbnd` with the syntax

```
[x,fval] = simulannealbnd(@objfun,x0)
```

The input arguments to `simulannealbnd` are

- `@objfun` — A function handle to the file that computes the objective function. "Compute Objective Functions" on page 2-2 explains how to write this file.
- `x0` — The initial guess of the optimal argument to the objective function.

The output arguments are

- `x` — The final point.
- `fval` — The value of the objective function at `x`.

For a description of additional input and output arguments, see the reference pages for `simulannealbnd`.

You can run the example described in "Minimize Function with Many Local Minima" on page 7-6 from the command line by entering

```
rng(10,'twister') % for reproducibility
[x,fval] = simulannealbnd(@dejong5fcn,[0 0])
```

This returns

```
Optimization terminated: change in best function value less than options.TolFun.

x =
  -16.1292  -15.8214
```

```
fval =
    6.9034
```

Additional Output Arguments

To get more information about the performance of the algorithm, you can call `simulannealbnd` with the syntax

```
[x,fval,exitflag,output] = simulannealbnd(@objfun,x0)
```

Besides `x` and `fval`, this function returns the following additional output arguments:

- `exitflag` — Flag indicating the reason the algorithm terminated
- `output` — Structure containing information about the performance of the algorithm

See the `simulannealbnd` reference pages for more information about these arguments.

Set Options for simulannealbnd at the Command Line

You can specify options by passing an options structure as an input argument to `simulannealbnd` using the syntax

```
[x,fval] = simulannealbnd(@objfun,x0,[],[],options)
```

This syntax does not specify any lower or upper bound constraints.

You create the options structure using the `saoptimset` function:

```
options = saoptimset('simulannealbnd')
```

This returns the structure `options` with the default values for its fields:

```
options =
          AnnealingFcn: @annealingfast
        TemperatureFcn: @temperatureexp
         AcceptanceFcn: @acceptancesa
                TolFun: 1.0000e-006
        StallIterLimit: '500*numberofvariables'
           MaxFunEvals: '3000*numberofvariables'
             TimeLimit: Inf
               MaxIter: Inf
        ObjectiveLimit: -Inf
```

```
             Display: 'final'
     DisplayInterval: 10
           HybridFcn: []
      HybridInterval: 'end'
            PlotFcns: []
        PlotInterval: 1
          OutputFcns: []
  InitialTemperature: 100
    ReannealInterval: 100
            DataType: 'double'
```

The value of each option is stored in a field of the options structure, such as `options.ReannealInterval`. You can display any of these values by entering `options` followed by the name of the field. For example, to display the interval for reannealing used for the simulated annealing algorithm, enter

```
options.ReannealInterval
ans =
   100
```

To create an options structure with a field value that is different from the default—for example, to set `ReannealInterval` to `300` instead of its default value `100`—enter

```
options = saoptimset('ReannealInterval',300)
```

This creates the options structure with all values set to their defaults, except for `ReannealInterval`, which is set to `300`.

If you now enter

```
simulannealbnd(@objfun,x0,[],[],options)
```

`simulannealbnd` runs the simulated annealing algorithm with a reannealing interval of `300`.

If you subsequently decide to change another field in the options structure, such as setting `PlotFcns` to `@saplotbestf`, which plots the best objective function value at each iteration, call `saoptimset` with the syntax

```
options = saoptimset(options,'PlotFcns',@saplotbestf)
```

This preserves the current values of all fields of `options` except for `PlotFcns`, which is changed to `@saplotbestf`. Note that if you omit the input argument `options`, `saoptimset` resets `ReannealInterval` to its default value `100`.

You can also set both `ReannealInterval` and `PlotFcns` with the single command

```
options = saoptimset('ReannealInterval',300, ...
                     'PlotFcns',@saplotbestf)
```

Reproduce Your Results

Because the simulated annealing algorithm is stochastic—that is, it makes random choices—you get slightly different results each time you run it. The algorithm uses the default MATLAB pseudorandom number stream. For more information about random number streams, see `RandStream`. Each time the algorithm calls the stream, its state changes. So the next time the algorithm calls the stream, it returns a different random number.

If you need to reproduce your results exactly, call `simulannealbnd` with the `output` argument. The `output` structure contains the current random number generator state in the `output.rngstate` field. Reset the state before running the function again.

For example, to reproduce the output of `simulannealbnd` applied to De Jong's fifth function, call `simulannealbnd` with the syntax

```
rng(10,'twister') % for reproducibility
[x,fval,exitflag,output] = simulannealbnd(@dejong5fcn,[0 0]);
```

Suppose the results are

```
x,fval

x =
  -16.1292  -15.8214

fval =
    6.9034
```

The state of the random number generator, `rngstate`, is stored in `output.rngstate`:

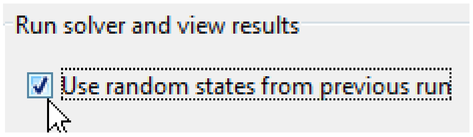
```
output.rngstate

ans =

    state: [625x1 uint32]
     type: 'mt19937ar'
```

Reset the stream by entering

```
stream = RandStream.getGlobalStream;
stream.State = output.rngstate.state;
```

If you now run `simulannealbnd` a second time, you get the same results.

You can reproduce your run in the Optimization app by checking the box **Use random states from previous run** in the **Run solver and view results** section.


Run solver and view results
Use random states from previous run

Note If you do not need to reproduce your results, it is better not to set the states of `RandStream`, so that you get the benefit of the randomness in these algorithms.

Minimization Using Simulated Annealing Algorithm

This example shows how to create and minimize an objective function using Simulated Annealing in the Global Optimization Toolbox.

A Simple Objective Function

We want to minimize a simple function of two variables

```
   min f(x) = (4 - 2.1*x1^2 + x1^4/3)*x1^2 + x1*x2 + (-4 + 4*x2^2)*x2^2;
    x
```

The above function is known as 'cam' as described in L.C.W. Dixon and G.P. Szego (eds.), Towards Global Optimisation 2, North-Holland, Amsterdam, 1978.

Coding the Objective Function

We create a MATLAB file named simple_objective.m with the following code in it:

```
   function y = simple_objective(x)
   y = (4 - 2.1*x(1)^2 + x(1)^4/3)*x(1)^2 + x(1)*x(2) + ...
       (-4 + 4*x(2)^2)*x(2)^2;
```

The Simulated Annealing solver assumes the objective function will take one input x where x has as many elements as the number of variables in the problem. The objective function computes the scalar value of the objective and returns it in its single return argument y.

Minimizing Using SIMULANNEALBND

To minimize our objective function using the SIMULANNEALBND function, we need to pass in a function handle to the objective function as well as specifying a start point as the second argument.

```
ObjectiveFunction = @simple_objective;
X0 = [0.5 0.5];   % Starting point
[x,fval,exitFlag,output] = simulannealbnd(ObjectiveFunction,X0)
```

```
Optimization terminated: change in best function value less than options.TolFun.

x =

   -0.0896    0.7130
```

```
fval =

   -1.0316


exitFlag =

     1


output =

     iterations: 2948
      funccount: 2971
        message: 'Optimization terminated: change in best function value l...'
       rngstate: [1x1 struct]
    problemtype: 'unconstrained'
    temperature: [2x1 double]
      totaltime: 4.3836
```

The first two output arguments returned by SIMULANNEALBND are `x`, the best point found, and `fval`, the function value at the best point. A third output argument, `exitFlag` returns a flag corresponding to the reason SIMULANNEALBND stopped. SIMULANNEALBND can also return a fourth argument, `output`, which contains information about the performance of the solver.

Bound Constrained Minimization

SIMULANNEALBND can be used to solve problems with bound constraints. The lower and upper bounds are passed to the solver as vectors. For each dimension i, the solver ensures that lb(i) <= x(i) <= ub(i), where x is a point selected by the solver during simulation. We impose the bounds on our problem by specifying a range -64 <= x(i) <= 64 for x(i).

```
lb = [-64 -64];
ub = [64 64];
```

Now, we can rerun the solver with lower and upper bounds as input arguments.

```
[x,fval,exitFlag,output] = simulannealbnd(ObjectiveFunction,X0,lb,ub);

fprintf('The number of iterations was : %d\n', output.iterations);
fprintf('The number of function evaluations was : %d\n', output.funccount);
fprintf('The best function value found was : %g\n', fval);
```

```
Optimization terminated: change in best function value less than options.TolFun.
The number of iterations was : 2428
The number of function evaluations was : 2447
The best function value found was : -1.03163
```

How Simulated Annealing Works

Simulated annealing mimics the annealing process to solve an optimization problem. It uses a temperature parameter that controls the search. The temperature parameter typically starts off high and is slowly "cooled" or lowered in every iteration. At each iteration a new point is generated and its distance from the current point is proportional to the temperature. If the new point has a better function value it replaces the current point and iteration counter is incremented. It is possible to accept and move forward with a worse point. The probability of doing so is directly dependent on the temperature. This unintuitive step sometime helps identify a new search region in hope of finding a better minimum.

An Objective Function with Additional Arguments

Sometimes we want our objective function to be parameterized by extra arguments that act as constants during the optimization. For example, in the previous objective function, say we want to replace the constants 4, 2.1, and 4 with parameters that we can change to create a family of objective functions. We can re-write the above function to take three additional parameters to give the new minimization problem.

```
min f(x) = (a - b*x1^2 + x1^4/3)*x1^2 + x1*x2 + (-c + c*x2^2)*x2^2;
 x
```

a, b, and c are parameters to the objective function that act as constants during the optimization (they are not varied as part of the minimization). One can create a MATLAB file called parameterized_objective.m containing the following code.

```
function y = parameterized_objective(x,a,b,c)
y = (a - b*x(1)^2 + x(1)^4/3)*x(1)^2 + x(1)*x(2) + ...
    (-c + c*x(2)^2)*x(2)^2;
```

Minimizing Using Additional Arguments

Again, we need to pass in a function handle to the objective function as well as a start point as the second argument.

SIMULANNEALBND will call our objective function with just one argument `x`, but our objective function has four arguments: x, a, b, c. We can use an anonymous function to capture the values of the additional arguments, the constants a, b, and c. We create a function handle 'ObjectiveFunction' to an anonymous function that takes one input `x`, but calls 'parameterized_objective' with x, a, b and c. The variables a, b, and c have values when the function handle 'ObjectiveFunction' is created, so these values are captured by the anonymous function.

```
a = 4; b = 2.1; c = 4;    % define constant values
ObjectiveFunction = @(x) parameterized_objective(x,a,b,c);
X0 = [0.5 0.5];
[x,fval] = simulannealbnd(ObjectiveFunction,X0)
```

```
Optimization terminated: change in best function value less than options.TolFun.

x =

    0.0898   -0.7127


fval =

   -1.0316
```

Multiobjective Optimization

What Is Multiobjective Optimization?

You might need to formulate problems with more than one objective, since a single objective with several constraints may not adequately represent the problem being faced. If so, there is a vector of objectives,
$F(x) = [F_1(x), F_2(x),...,F_m(x)]$,
that must be traded off in some way. The relative importance of these objectives is not generally known until the system's best capabilities are determined and tradeoffs between the objectives fully understood. As the number of objectives increases, tradeoffs are likely to become complex and less easily quantified. The designer must rely on his or her intuition and ability to express preferences throughout the optimization cycle. Thus, requirements for a multiobjective design strategy must enable a natural problem formulation to be expressed, and be able to solve the problem and enter preferences into a numerically tractable and realistic design problem.

Multiobjective optimization is concerned with the minimization of a vector of objectives $F(x)$ that can be the subject of a number of constraints or bounds:

$$\min_{x \in \mathbf{R}^n} F(x), \text{ subject to}$$
$$G_i(x) = 0,\ i = 1,...,k_e;\ G_i(x) \le 0,\ i = k_e + 1,...,k;\ l \le x \le u.$$

Note that because $F(x)$ is a vector, if any of the components of $F(x)$ are competing, there is no unique solution to this problem. Instead, the concept of noninferiority in Zadeh [4] (also called Pareto optimality in Censor [1] and Da Cunha and Polak [2]) must be used to characterize the objectives. A noninferior solution is one in which an improvement in one objective requires a degradation of another. To define this concept more precisely, consider a feasible region, Ω, in the parameter space. x is an element of the n-dimensional real numbers $x \in \mathbf{R}^n$ that satisfies all the constraints, i.e.,

$$\Omega = \left\{x \in \mathbf{R}^n\right\},$$

subject to

$$G_i(x) = 0,\ i = 1,...,k_e,$$
$$G_i(x) \le 0,\ i = k_e + 1,...,k,$$
$$l \le x \le u.$$

This allows definition of the corresponding feasible region for the objective function space Λ:

$$\Lambda = \left\{ y \in \mathbf{R}^m : y = F(x), x \in \Omega \right\}.$$

The performance vector $F(x)$ maps parameter space into objective function space, as represented in two dimensions in the figure Mapping from Parameter Space into Objective Function Space.

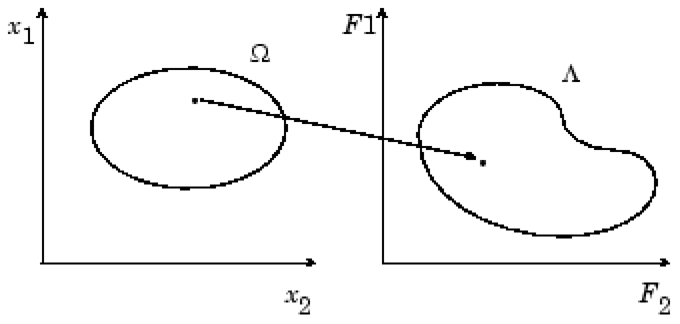


Figure 8-1. Mapping from Parameter Space into Objective Function Space

A noninferior solution point can now be defined.

Definition: Point $x^* \in \Omega$ is a noninferior solution if for some neighborhood of x^* there does not exist a Δx such that $(x^* + \Delta x) \in \Omega$ and

$$F_i\left(x^* + \Delta x\right) \le F_i(x^*),\ i = 1,...,m,\ \text{and}$$
$$F_j\left(x^* + \Delta x\right) < F_j(x^*) \text{ for at least one } j.$$

In the two-dimensional representation of the figure Set of Noninferior Solutions, the set of noninferior solutions lies on the curve between C and D. Points A and B represent specific noninferior points.

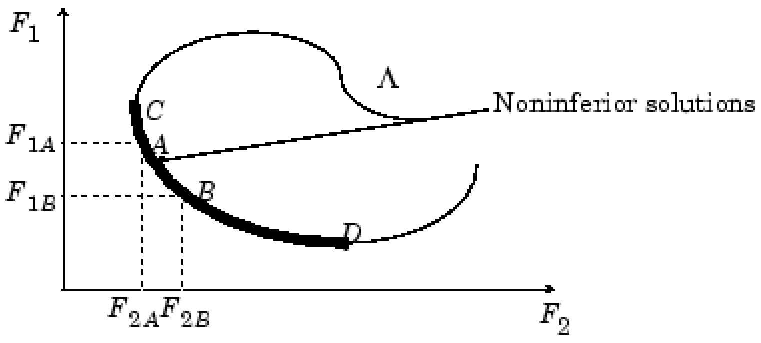


Figure 8-2. Set of Noninferior Solutions

A and B are clearly noninferior solution points because an improvement in one objective, F_1, requires a degradation in the other objective, F_2, i.e., $F_{1B} < F_{1A}$, $F_{2B} > F_{2A}$.

Since any point in Ω that is an inferior point represents a point in which improvement can be attained in all the objectives, it is clear that such a point is of no value. Multiobjective optimization is, therefore, concerned with the generation and selection of noninferior solution points.

Noninferior solutions are also called *Pareto optima*. A general goal in multiobjective optimization is constructing the Pareto optima. The algorithm used in `gamultiobj` is described in Deb [3].

Use gamultiobj

In this section...

Problem Formulation

The `gamultiobj` solver attempts to create a set of Pareto optima for a multiobjective minimization. You may optionally set bounds or other constraints on variables. `gamultiobj` uses the genetic algorithm for finding local Pareto optima. As in the `ga` function, you may specify an initial population, or have the solver generate one automatically.

The fitness function for use in `gamultiobj` should return a vector of type `double`. The population may be of type `double`, a bit string vector, or can be a custom-typed vector. As in `ga`, if you use a custom population type, you must write your own creation, mutation, and crossover functions that accept inputs of that population type, and specify these functions in the following fields, respectively:

- **Creation function** (`CreationFcn`)
- **Mutation function** (`MutationFcn`)
- **Crossover function** (`CrossoverFcn`)

You can set the initial population in a variety of ways. Suppose that you choose a population of size m. (The default population size is 15 times the number of variables n.) You can set the population:

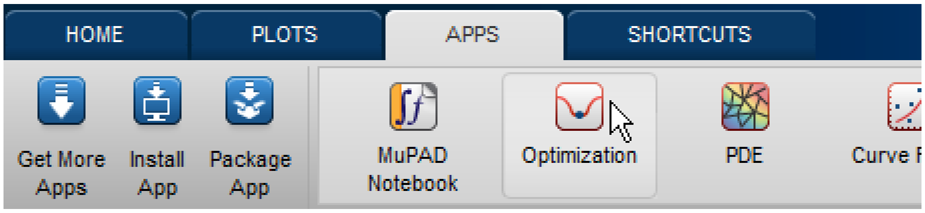
- As an m-by-n matrix, where the rows represent m individuals.
- As a k-by-n matrix, where $k < m$. The remaining $m - k$ individuals are generated by a creation function.
- The entire population can be created by a creation function.

Use gamultiobj with the Optimization app

You can access `gamultiobj` from the Optimization app. Enter

```
optimtool('gamultiobj')
```

at the command line, or enter `optimtool` and then choose `gamultiobj` from the **Solver** menu. You can also start the tool from the MATLAB **Apps** tab.



If the **Quick Reference** help pane is closed, you can open it by clicking the ">>" button on the upper right: . The help pane briefly explains all the available options.

You can create an options structure in the Optimization app, export it to the MATLAB workspace, and use the structure at the command line. For details, see "Importing and Exporting Your Work" in the Optimization Toolbox documentation.

Multiobjective Optimization with Two Objectives

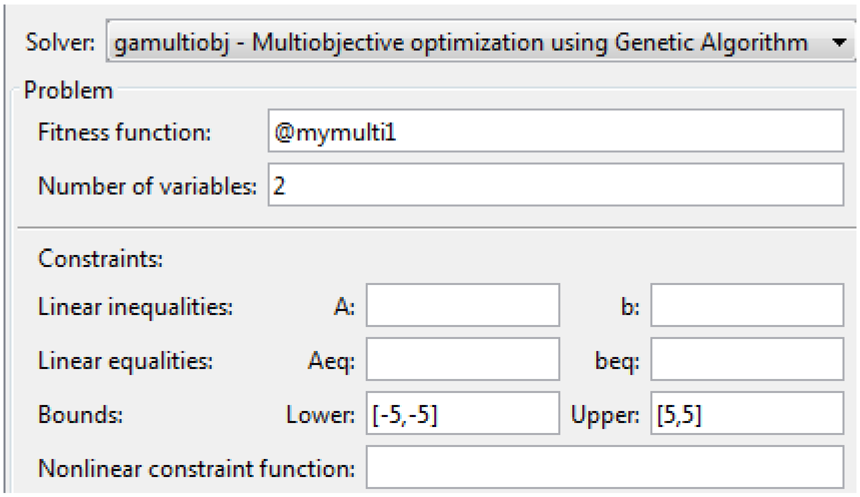
This example has a two-objective fitness function *f*(*x*), where *x* is also two-dimensional:

```
function f = mymulti1(x)

f(1) = x(1)^4 - 10*x(1)^2+x(1)*x(2) + x(2)^4 -(x(1)^2)*(x(2)^2);
f(2) = x(2)^4 - (x(1)^2)*(x(2)^2) + x(1)^4 + x(1)*x(2);
```

Create this function file before proceeding.

Performing the Optimization with Optimization App

1 To define the optimization problem, start the Optimization app, and set it as pictured.



Solver: gamultiobj - Multiobjective optimization using Genetic Algorithm

Problem

Fitness function: @mymulti1

Number of variables: 2

Constraints:

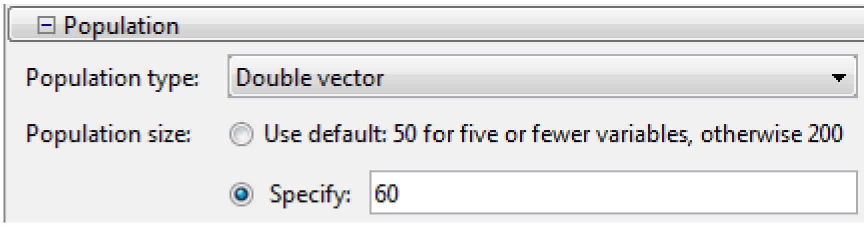
Linear inequalities: A: b:

Linear equalities: Aeq: beq:

Bounds: Lower: [-5,-5] Upper: [5,5]

Nonlinear constraint function:

2 Set the options for the problem as pictured.

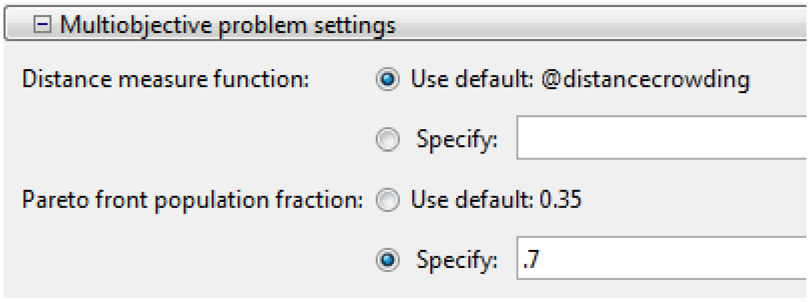


Population

Population type: Double vector

Population size: ○ Use default: 50 for five or fewer variables, otherwise 200

[✓ Specify: 60]



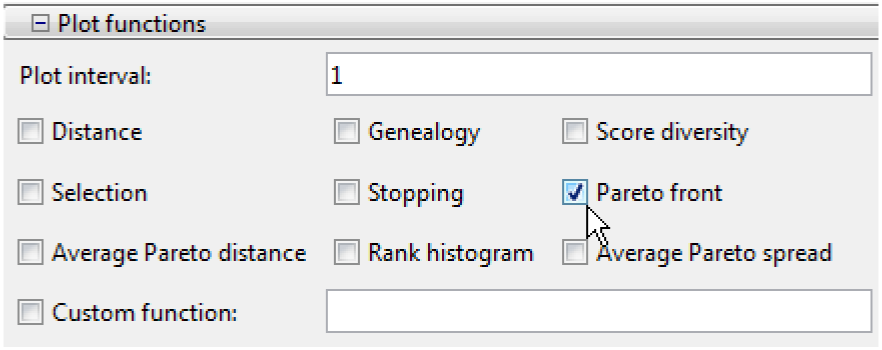
Multiobjective problem settings

Distance measure function: [✓ Use default: @distancecrowding]

○ Specify:

Pareto front population fraction: ○ Use default: 0.35

[✓ Specify: .7]

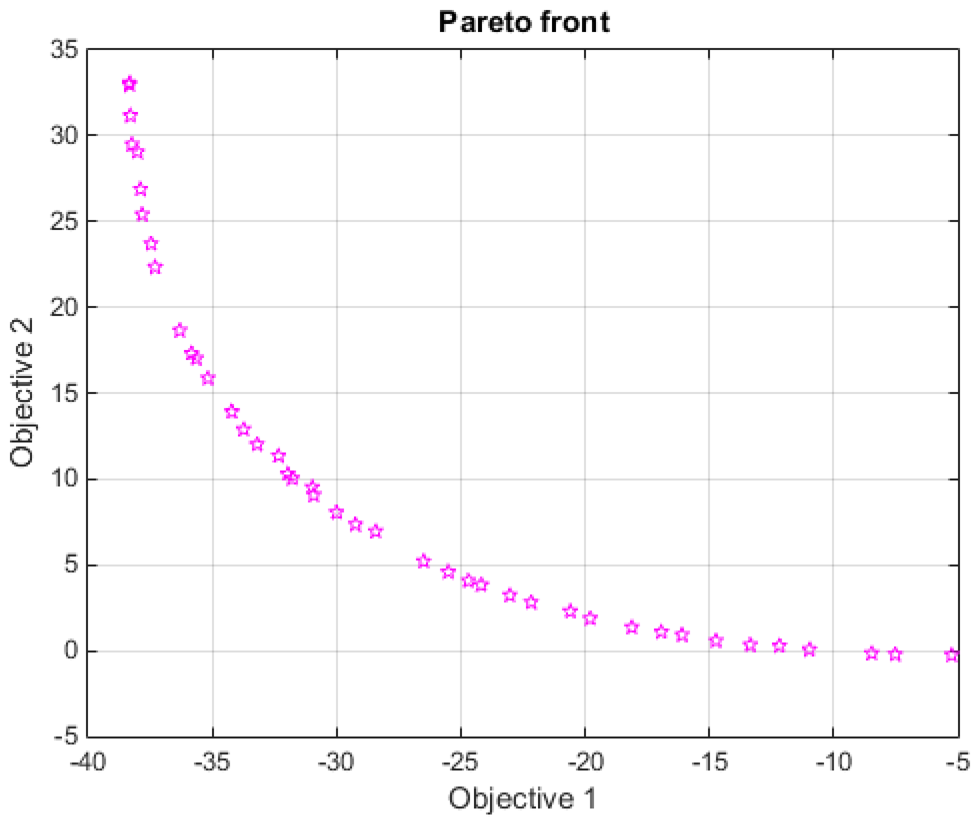


Plot interval: 1

☐ Distance	☐ Genealogy	☐ Score diversity
☐ Selection	☐ Stopping	☑ Pareto front
☐ Average Pareto distance	☐ Rank histogram	☐ Average Pareto spread
☐ Custom function:		

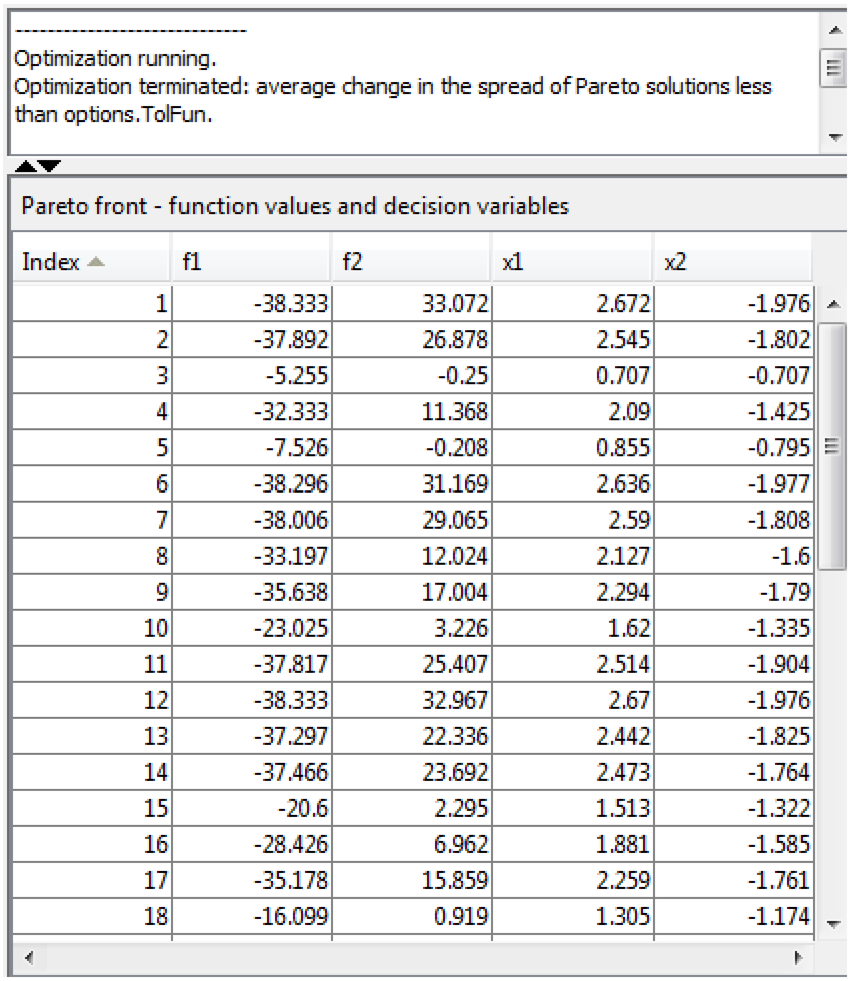
3 Run the optimization by clicking **Start** under **Run solver and view results**.

A plot appears in a figure window.



This plot shows the tradeoff between the two components of *f*. It is plotted in objective function space; see the figure Set of Noninferior Solutions.

The results of the optimization appear in the following table containing both objective function values and the value of the variables.

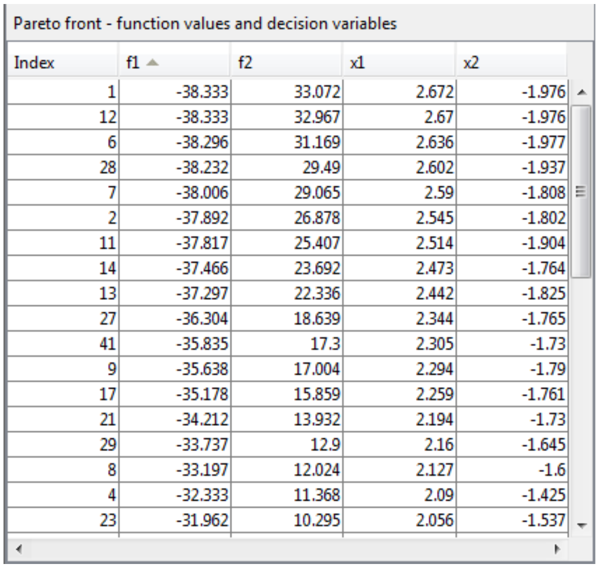


```
------------------------------
Optimization running.
Optimization terminated: average change in the spread of Pareto solutions less
than options.TolFun.
```

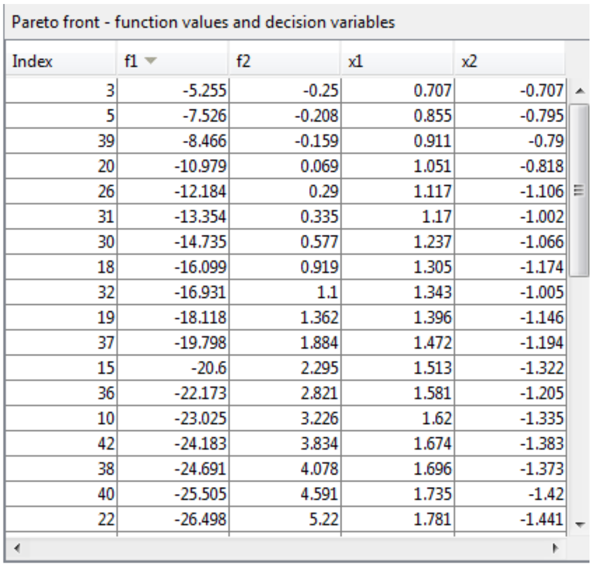
Pareto front - function values and decision variables

Index	f1	f2	x1	x2
1	-38.333	33.072	2.672	-1.976
2	-37.892	26.878	2.545	-1.802
3	-5.255	-0.25	0.707	-0.707
4	-32.333	11.368	2.09	-1.425
5	-7.526	-0.208	0.855	-0.795
6	-38.296	31.169	2.636	-1.977
7	-38.006	29.065	2.59	-1.808
8	-33.197	12.024	2.127	-1.6
9	-35.638	17.004	2.294	-1.79
10	-23.025	3.226	1.62	-1.335
11	-37.817	25.407	2.514	-1.904
12	-38.333	32.967	2.67	-1.976
13	-37.297	22.336	2.442	-1.825
14	-37.466	23.692	2.473	-1.764
15	-20.6	2.295	1.513	-1.322
16	-28.426	6.962	1.881	-1.585
17	-35.178	15.859	2.259	-1.761
18	-16.099	0.919	1.305	-1.174

You can sort the table by clicking a heading. Click the heading again to sort it in the reverse order. The following figures show the result of clicking the heading `f1`.


Pareto front - function values and decision variables

Index	f1 ▲	f2	x1	x2
1	-38.333	33.072	2.672	-1.976
12	-38.333	32.967	2.67	-1.976
6	-38.296	31.169	2.636	-1.977
28	-38.232	29.49	2.602	-1.937
7	-38.006	29.065	2.59	-1.808
2	-37.892	26.878	2.545	-1.802
11	-37.817	25.407	2.514	-1.904
14	-37.466	23.692	2.473	-1.764
13	-37.297	22.336	2.442	-1.825
27	-36.304	18.639	2.344	-1.765
41	-35.835	17.3	2.305	-1.73
9	-35.638	17.004	2.294	-1.79
17	-35.178	15.859	2.259	-1.761
21	-34.212	13.932	2.194	-1.73
29	-33.737	12.9	2.16	-1.645
8	-33.197	12.024	2.127	-1.6
4	-32.333	11.368	2.09	-1.425
23	-31.962	10.295	2.056	-1.537


Pareto front - function values and decision variables

Index	f1 ▼	f2	x1	x2
3	-5.255	-0.25	0.707	-0.707
5	-7.526	-0.208	0.855	-0.795
39	-8.466	-0.159	0.911	-0.79
20	-10.979	0.069	1.051	-0.818
26	-12.184	0.29	1.117	-1.106
31	-13.354	0.335	1.17	-1.002
30	-14.735	0.577	1.237	-1.066
18	-16.099	0.919	1.305	-1.174
32	-16.931	1.1	1.343	-1.005
19	-18.118	1.362	1.396	-1.146
37	-19.798	1.884	1.472	-1.194
15	-20.6	2.295	1.513	-1.322
36	-22.173	2.821	1.581	-1.205
10	-23.025	3.226	1.62	-1.335
42	-24.183	3.834	1.674	-1.383
38	-24.691	4.078	1.696	-1.373
40	-25.505	4.591	1.735	-1.42
22	-26.498	5.22	1.781	-1.441

Performing the Optimization at the Command Line

To perform the same optimization at the command line:

1 Set the options:

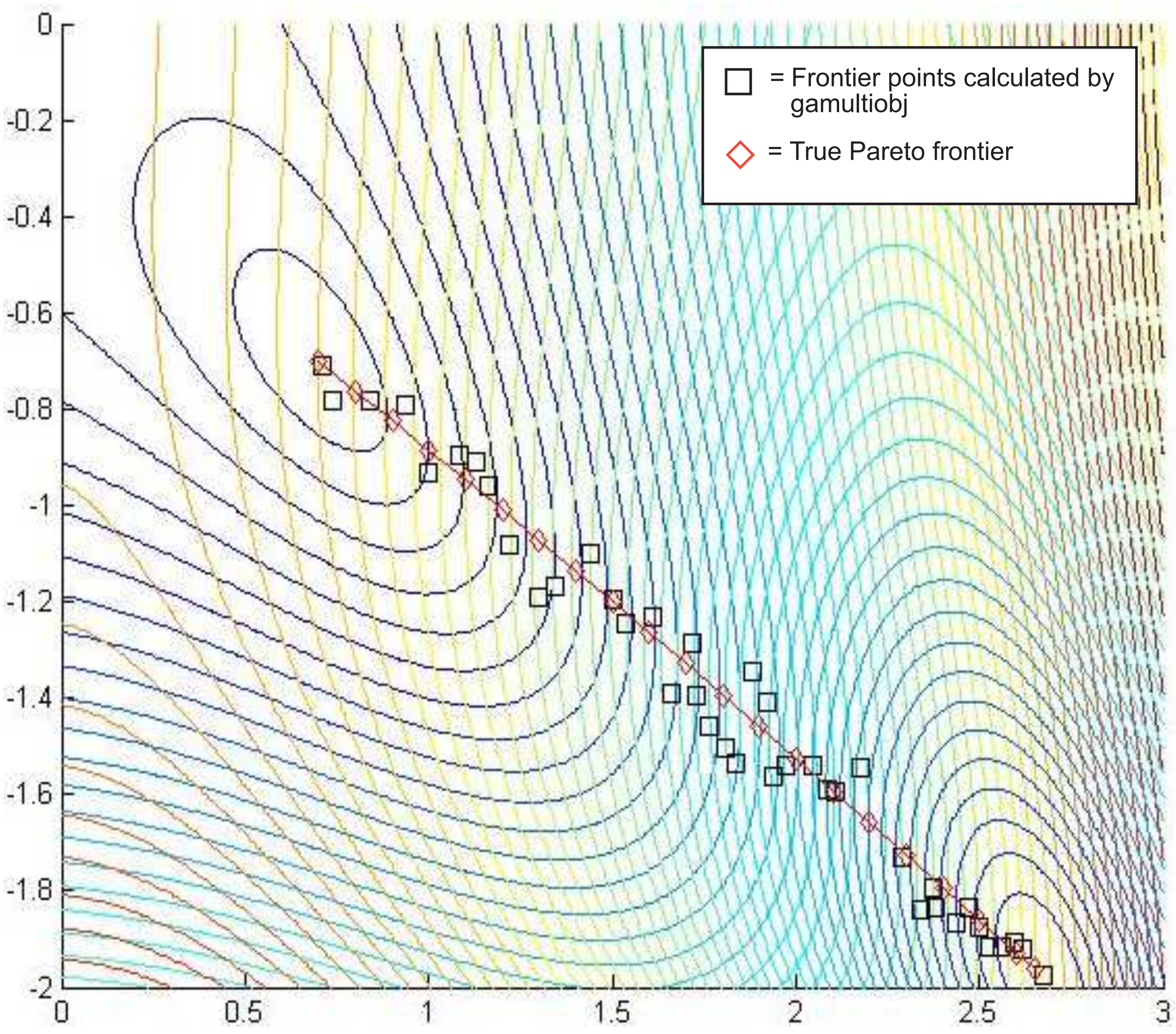
```
options = gaoptimset('PopulationSize',60,...
          'ParetoFraction',0.7,'PlotFcns',@gaplotpareto);
```

2 Run the optimization using the options:

```
[x,fval,flag,output,population] = gamultiobj(@mymulti1,2,...
                           [],[],[],[],[-5,-5],[5,5],options);
```

Alternate Views

There are other ways of regarding the problem. The following figure contains a plot of the level curves of the two objective functions, the Pareto frontier calculated by `gamultiobj` (boxes), and the x-values of the true Pareto frontier (diamonds connected by a nearly-straight line). The true Pareto frontier points are where the level curves of the objective functions are parallel. They were calculated by finding where the gradients of the objective functions are parallel. The figure is plotted in parameter space; see the figure Mapping from Parameter Space into Objective Function Space.



Contours of objective functions, and Pareto frontier

`gamultiobj` found the ends of the line segment, meaning it found the full extent of the Pareto frontier.

Options and Syntax: Differences from ga

The syntax and options for `gamultiobj` are similar to those for `ga`, with the following differences:

- `gamultiobj` uses only the `'penalty'` algorithm for nonlinear constraints. See "Nonlinear Constraint Solver Algorithms" on page 5-49.
- `gamultiobj` takes an option `DistanceMeasureFcn`, a function that assigns a distance measure to each individual with respect to its neighbors.
- `gamultiobj` takes an option `ParetoFraction`, a number between 0 and 1 that specifies the fraction of the population on the best Pareto frontier to be kept during the optimization. If there is only one Pareto frontier, this option is ignored.
- `gamultiobj` uses only the `Tournament` selection function.
- `gamultiobj` uses elite individuals differently than `ga`. It sorts noninferior individuals above inferior ones, so it uses elite individuals automatically.
- `gamultiobj` has only one hybrid function, `fgoalattain`.
- `gamultiobj` does not have a stall time limit.
- `gamultiobj` has different plot functions available.
- `gamultiobj` does not have a choice of scaling function.

Bibliography

[1] Censor, Y., “Pareto Optimality in Multiobjective Problems,” *Appl. Math. Optimiz.*, Vol. 4, pp 41–59, 1977.

[2] Da Cunha, N.O. and E. Polak, “Constrained Minimization Under Vector-Valued Criteria in Finite Dimensional Spaces,” *J. Math. Anal. Appl.*, Vol. 19, pp 103–124, 1967.

[3] Deb, Kalyanmoy, “Multi-Objective Optimization using Evolutionary Algorithms,” John Wiley & Sons, Ltd, Chichester, England, 2001.

[4] Zadeh, L.A., “Optimality and Nonscalar-Valued Performance Criteria,” *IEEE Trans. Automat. Contr.*, Vol. AC-8, p. 1, 1963.

Parallel Processing

Background

In this section...
"Parallel Processing Types in Global Optimization Toolbox" on page 9-2
"How Toolbox Functions Distribute Processes" on page 9-3

Parallel Processing Types in Global Optimization Toolbox

Parallel processing is an attractive way to speed optimization algorithms. To use parallel processing, you must have a Parallel Computing Toolbox license, and have a parallel worker pool (`parpool`). For more information, see "How to Use Parallel Processing" on page 9-12.

Global Optimization Toolbox solvers use parallel computing in various ways.

Solver	Parallel?	Parallel Characteristics
`GlobalSearch`	×	No parallel functionality. However, `fmincon` can use parallel gradient estimation when run in `GlobalSearch`. See "Using Parallel Computing in Optimization Toolbox" in the Optimization Toolbox documentation.
`MultiStart`	✓	Start points distributed to multiple processors. From these points, local solvers run to completion. For more details, see "MultiStart" on page 9-5 and "How to Use Parallel Processing" on page 9-12.
		For `fmincon`, no parallel gradient estimation with parallel `MultiStart`.
`ga`, `gamultiobj`	✓	Population evaluated in parallel, which occurs once per iteration. For more details, see "Genetic Algorithm" on page 9-8 and "How to Use Parallel Processing" on page 9-12.
		No vectorization of fitness or constraint functions.
`particleswarm`	✓	Population evaluated in parallel, which occurs once per iteration. For more details, see "Particle Swarm" on page 9-10 and "How to Use Parallel Processing" on page 9-12.
		No vectorization of fitness or constraint functions.

Solver	Parallel?	Parallel Characteristics
`patternsearch`	✓	Poll points evaluated in parallel, which occurs once per iteration. For more details, see “Pattern Search” on page 9-6 and “How to Use Parallel Processing” on page 9-12.
		No vectorization of fitness or constraint functions.
`simulannealbnd`	×	No parallel functionality. However, `simulannealbnd` can use a hybrid function that runs in parallel. See “Simulated Annealing” on page 9-11.

In addition, several solvers have hybrid functions that run after they finish. Some hybrid functions can run in parallel. Also, most `patternsearch` search methods can run in parallel. For more information, see “Parallel Search Functions or Hybrid Functions” on page 9-15.

How Toolbox Functions Distribute Processes

- “parfor Characteristics and Caveats” on page 9-3
- “MultiStart” on page 9-5
- “GlobalSearch” on page 9-6
- “Pattern Search” on page 9-6
- “Genetic Algorithm” on page 9-8
- “Parallel Computing with gamultiobj” on page 9-9
- “Particle Swarm” on page 9-10
- “Simulated Annealing” on page 9-11

parfor Characteristics and Caveats

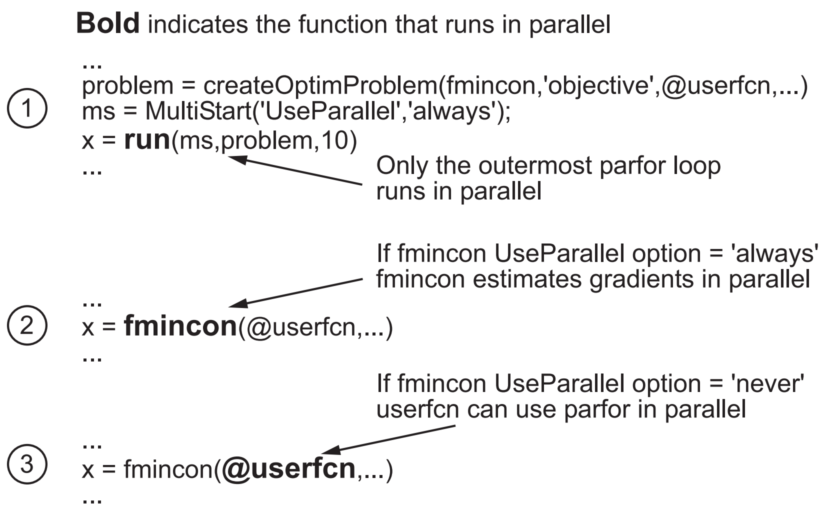
No Nested parfor Loops

Solvers employ the Parallel Computing Toolbox `parfor` function to perform parallel computations.

Note: `parfor` does not work in parallel when called from within another `parfor` loop.

Suppose, for example, your objective function `userfcn` calls `parfor`, and you want to call `fmincon` using `MultiStart` and parallel processing. Suppose also that the conditions for parallel gradient evaluation of `fmincon` are satisfied, as given in “Parallel Optimization Functionality”. The figure When parfor Runs In Parallel shows three cases:

1. The outermost loop is parallel `MultiStart`. Only that loop runs in parallel.
2. The outermost `parfor` loop is in `fmincon`. Only `fmincon` runs in parallel.
3. The outermost `parfor` loop is in `userfcn`. In this case, `userfcn` can use `parfor` in parallel.


Bold indicates the function that runs in parallel
1
...
problem = createOptimProblem(fmincon,'objective',@userfcn,...)
ms = MultiStart('UseParallel','always');
x = **run**(ms,problem,10)
...
Only the outermost parfor loop runs in parallel
If fmincon UseParallel option = 'always' fmincon estimates gradients in parallel
2
...
x = **fmincon**(@userfcn,...)
...
If fmincon UseParallel option = 'never' userfcn can use parfor in parallel
3
...
x = fmincon(**@userfcn**,...)
...

When parfor Runs In Parallel

Parallel Random Numbers Are Not Reproducible

Random number sequences in MATLAB are pseudorandom, determined from a *seed*, or an initial setting. Parallel computations use seeds that are not necessarily controllable or reproducible. For example, each instance of MATLAB has a default global setting that determines the current seed for random sequences.

For `patternsearch`, if you select MADS as a poll or search method, parallel pattern search does not have reproducible runs. If you select the genetic algorithm or Latin hypercube as search methods, parallel pattern search does not have reproducible runs.

For `ga` and `gamultiobj`, parallel population generation gives nonreproducible results.

`MultiStart` is different. You *can* have reproducible runs from parallel `MultiStart`. Runs are reproducible because `MultiStart` generates pseudorandom start points locally, and then distributes the start points to parallel processors. Therefore, the parallel processors do not use random numbers. For more details, see "Parallel Processing and Random Number Streams" on page 3-71.

Limitations and Performance Considerations

More caveats related to `parfor` appear in the "parfor Limitations" section of the Parallel Computing Toolbox documentation.

For information on factors that affect the speed of parallel computations, and factors that affect the results of parallel computations, see "Improving Performance with Parallel Computing" in the Optimization Toolbox documentation. The same considerations apply to parallel computing with Global Optimization Toolbox functions.

MultiStart

`MultiStart` can automatically distribute a problem and start points to multiple processes or processors. The problems run independently, and `MultiStart` combines the distinct local minima into a vector of `GlobalOptimSolution` objects. `MultiStart` uses parallel computing when you:

- Have a license for Parallel Computing Toolbox software.
- Enable parallel computing with `parpool`, a Parallel Computing Toolbox function.
- Set the `UseParallel` property to `true` in the `MultiStart` object:

  ```
  ms = MultiStart('UseParallel',true);
  ```

When these conditions hold, `MultiStart` distributes a problem and start points to processes or processors one at a time. The algorithm halts when it reaches a stopping condition or runs out of start points to distribute. If the `MultiStart` `Display` property is `'iter'`, then `MultiStart` displays:

```
Running the local solvers in parallel.
```

For an example of parallel `MultiStart`, see "Parallel MultiStart" on page 3-89.

Implementation Issues in Parallel MultiStart

`fmincon` cannot estimate gradients in parallel when used with parallel `MultiStart`. This lack of parallel gradient estimation is due to the limitation of `parfor` described in "No Nested parfor Loops" on page 9-3.

`fmincon` can take longer to estimate gradients in parallel rather than in serial. In this case, using `MultiStart` with parallel gradient estimation in `fmincon` amplifies the slowdown. For example, suppose the `ms` `MultiStart` object has `UseParallel` set to `false`. Suppose `fmincon` takes 1 s longer to solve `problem` with `problem.options.UseParallel` set to `true`. Then `run(ms,problem,200)` takes 200 s longer than the same run with `problem.options.UseParallel` set to `false`

Note: When executing serially, `parfor` loops run slower than `for` loops. Therefore, for best performance, set your local solver `UseParallel` option to `false` when the `MultiStart` `UseParallel` property is `true`.

Note: Even when running in parallel, a solver occasionally calls the objective and nonlinear constraint functions serially on the host machine. Therefore, ensure that your functions have no assumptions about whether they are evaluated in serial and parallel.

GlobalSearch

`GlobalSearch` does not distribute a problem and start points to multiple processes or processors. However, when `GlobalSearch` runs the `fmincon` local solver, `fmincon` can estimate gradients by parallel finite differences. `fmincon` uses parallel computing when you:

- Have a license for Parallel Computing Toolbox software.
- Enable parallel computing with `parpool`, a Parallel Computing Toolbox function.
- Set the `UseParallel` option to `true` with `optimoptions`. Set this option in the `problem` structure:

  ```
  opts = optimoptions(@fmincon,'UseParallel',true,'Algorithm','sqp');
  problem = createOptimProblem('fmincon','objective',@myobj,...
      'x0',startpt,'options',opts);
  ```

For more details, see “Using Parallel Computing in Optimization Toolbox” in the Optimization Toolbox documentation.

Pattern Search

`patternsearch` can automatically distribute the evaluation of objective and constraint functions associated with the points in a pattern to multiple processes or processors. `patternsearch` uses parallel computing when you:

- Have a license for Parallel Computing Toolbox software.

- Enable parallel computing with `parpool`, a Parallel Computing Toolbox function.
- Set the following options using `psoptimset` or the Optimization app:
  - `CompletePoll` is `'on'`.
  - `Vectorized` is `'off'` (default).
  - `UseParallel` is `true`.

When these conditions hold, the solver computes the objective function and constraint values of the pattern search in parallel during a poll. Furthermore, `patternsearch` overrides the setting of the `Cache` option, and uses the default `'off'` setting.

Note: Even when running in parallel, `patternsearch` occasionally calls the objective and nonlinear constraint functions serially on the host machine. Therefore, ensure that your functions have no assumptions about whether they are evaluated in serial or parallel.

Parallel Search Function

`patternsearch` can optionally call a search function at each iteration. The search is parallel when you:

- Set `CompleteSearch` to `'on'`.
- Do not set the search method to `@searchneldermead` or `custom`.
- Set the search method to a `patternsearch` poll method or Latin hypercube search, and set `UseParallel` to `true`.
- Or, if you set the search method to `ga`, create a search method option structure with `UseParallel` set to `true`.

Implementation Issues in Parallel Pattern Search

The limitations on `patternsearch` options, listed in “Pattern Search” on page 9-6, arise partly from the limitations of `parfor`, and partly from the nature of parallel processing:

- `Cache` is overridden to be `'off'` — `patternsearch` implements `Cache` as a persistent variable. `parfor` does not handle persistent variables, because the variable could have different settings at different processors.

- `CompletePoll` is `'on'` — `CompletePoll` determines whether a poll stops as soon as `patternsearch` finds a better point. When searching in parallel, `parfor` schedules all evaluations simultaneously, and `patternsearch` continues after all evaluations complete. `patternsearch` cannot halt evaluations after they start.
- `Vectorized` is `'off'` — `Vectorized` determines whether `patternsearch` evaluates all points in a pattern with one function call in a vectorized fashion. If `Vectorized` is `'on'`, `patternsearch` does not distribute the evaluation of the function, so does not use `parfor`.

Genetic Algorithm

`ga` and `gamultiobj` can automatically distribute the evaluation of objective and nonlinear constraint functions associated with a population to multiple processors. `ga` uses parallel computing when you:

- Have a license for Parallel Computing Toolbox software.
- Enable parallel computing with `parpool`, a Parallel Computing Toolbox function.
- Set the following options using `gaoptimset` or the Optimization app:
  - `Vectorized` is `'off'` (default).
  - `UseParallel` is `true`.

When these conditions hold, `ga` computes the objective function and nonlinear constraint values of the individuals in a population in parallel.

Note: Even when running in parallel, `ga` occasionally calls the fitness and nonlinear constraint functions serially on the host machine. Therefore, ensure that your functions have no assumptions about whether they are evaluated in serial or parallel.

Implementation Issues in Parallel Genetic Algorithm

The limitations on options, listed in “Genetic Algorithm” on page 9-8, arise partly from limitations of `parfor`, and partly from the nature of parallel processing:

- `Vectorized` is `'off'` — `Vectorized` determines whether `ga` evaluates an entire population with one function call in a vectorized fashion. If `Vectorized` is `'on'`, `ga` does not distribute the evaluation of the function, so does not use `parfor`.

ga can have a hybrid function that runs after it finishes; see “Include a Hybrid Function” on page 5-104. If you want the hybrid function to take advantage of parallel computation, set its options separately so that UseParallel is true. If the hybrid function is patternsearch, set CompletePoll to 'on' so that patternsearch runs in parallel.

If the hybrid function is fmincon, set the following options with optimoptions to have parallel gradient estimation:

- GradObj must not be 'on' — it can be 'off' or [].
- Or, if there is a nonlinear constraint function, GradConstr must not be 'on' — it can be 'off' or [].

To find out how to write options for the hybrid function, see “Parallel Hybrid Functions” on page 9-17.

Parallel Computing with gamultiobj

Parallel computing with gamultiobj works almost the same as with ga. For detailed information, see “Genetic Algorithm” on page 9-8.

The difference between parallel computing with gamultiobj and ga has to do with the hybrid function. gamultiobj allows only one hybrid function, fgoalattain. This function optionally runs after gamultiobj finishes its run. Each individual in the calculated Pareto frontier, that is, the final population found by gamultiobj, becomes the starting point for an optimization using fgoalattain. These optimizations run in parallel. The number of processors performing these optimizations is the smaller of the number of individuals and the size of your parpool.

For fgoalattain to run in parallel, set its options correctly:

```
fgoalopts = optimoptions(@fgoalattain,'UseParallel',true)
gaoptions = gaoptimset('HybridFcn',{@fgoalattain,fgoalopts});
```

Run gamultiobj with gaoptions, and fgoalattain runs in parallel. For more information about setting the hybrid function, see “Hybrid Function Options” on page 10-48.

gamultiobj calls fgoalattain using a parfor loop, so fgoalattain does not estimate gradients in parallel when used as a hybrid function with gamultiobj. For more information, see “No Nested parfor Loops” on page 9-3.

Particle Swarm

`particleswarm` can automatically distribute the evaluation of the objective function associated with a population to multiple processors. `particleswarm` uses parallel computing when you:

- Have a license for Parallel Computing Toolbox software.
- Enable parallel computing with `parpool`, a Parallel Computing Toolbox function.
- Set the following options using `optimoptions`:
  - `Vectorized` is `'off'` (default).
  - `UseParallel` is `true`.

When these conditions hold, `particleswarm` computes the objective function of the particles in a population in parallel.

Note: Even when running in parallel, `particleswarm` occasionally calls the objective function serially on the host machine. Therefore, ensure that your objective function has no assumptions about whether it is evaluated in serial or parallel.

Implementation Issues in Parallel Particle Swarm Optimization

The limitations on options, listed in "Particle Swarm" on page 9-10, arise partly from limitations of `parfor`, and partly from the nature of parallel processing:

- `Vectorized` is `'off'` — `Vectorized` determines whether `particleswarm` evaluates an entire population with one function call in a vectorized fashion. If `Vectorized` is `'on'`, `particleswarm` does not distribute the evaluation of the function, so does not use `parfor`.

`particleswarm` can have a hybrid function that runs after it finishes; see "Include a Hybrid Function" on page 5-104. If you want the hybrid function to take advantage of parallel computation, set its options separately so that `UseParallel` is `true`. If the hybrid function is `patternsearch`, set `CompletePoll` to `'on'` so that `patternsearch` runs in parallel.

If the hybrid function is `fmincon`, set the `GradObj` option to `'off'` or `[]` with `optimoptions` to have parallel gradient estimation.

To find out how to write options for the hybrid function, see "Parallel Hybrid Functions" on page 9-17.

Simulated Annealing

`simulannealbnd` does not run in parallel automatically. However, it can call hybrid functions that take advantage of parallel computing. To find out how to write options for the hybrid function, see "Parallel Hybrid Functions" on page 9-17.

How to Use Parallel Processing

In this section...

Multicore Processors

If you have a multicore processor, you might see speedup using parallel processing. You can establish a parallel pool of several workers with a Parallel Computing Toolbox license. For a description of Parallel Computing Toolbox software, see "Getting Started with Parallel Computing Toolbox".

Suppose you have a dual-core processor, and want to use parallel computing:

- Enter

  ```
  parpool
  ```
  at the command line. MATLAB starts a pool of workers using the multicore processor. If you had previously set a nondefault cluster profile, you can enforce multicore (local) computing:

  ```
  parpool('local')
  ```

  Note: Depending on your preferences, MATLAB can start a parallel pool automatically. To enable this feature, check **Automatically create a parallel pool** in **Home > Parallel > Parallel Preferences**.

- Set your solver to use parallel processing.

MultiStart	patternsearch	ga	particleswarm

`ms =`
`MultiStart('UsePara`
`true);`
 	
or
	`options =`
	`psoptimset('UsePara`
	`true,`
	`'CompletePoll',`
	`'on',`
		`options =`
		`gaoptimset('UsePara`
		`true,`
		`'Vectorized',`
		`'off');`
			`options =`
			`optimoptions('particleswarm'`
			`'UseParallel',`
			`true,`
			`'Vectorized',`
			`'off');`

MultiStart	patternsearch	ga	particleswarm
`ms.UseParallel = true`	`'Vectorized', 'off');`		
	For Optimization app: • **Options > User function evaluation > Evaluate objective and constraint functions > in parallel** • **Options > Complete poll > on**	For Optimization app: • **Options > User function evaluation > Evaluate fitness and constraint functions > in parallel**	

When you run an applicable solver with `options`, applicable solvers automatically use parallel computing.

To stop computing optimizations in parallel, set `UseParallel` to `false`, or set the Optimization app not to compute in parallel. To halt all parallel computation, enter

```
delete(gcp)
```

Processor Network

If you have multiple processors on a network, use Parallel Computing Toolbox functions and MATLAB Distributed Computing Server™ software to establish parallel computation. Here are the steps to take:

1 Make sure your system is configured properly for parallel computing. Check with your systems administrator, or refer to the Parallel Computing Toolbox documentation.

   To perform a basic check:

   a At the command line, enter

      ```
      parpool(prof)
      ```

where `prof` is your cluster profile.

b Workers must be able to access your objective function file and, if applicable, your nonlinear constraint function file. There are two ways of ensuring access:

i Distribute the files to the workers using the `parpool` `AttachedFiles` argument. For example, if `objfun.m` is your objective function file, and `constrfun.m` is your nonlinear constraint function file, enter

```
parpool('AttachedFiles',{'objfun.m','constrfun.m'});
```

Workers access their own copies of the files.

ii Give a network file path to your files. If *`network_file_path`* is the network path to your objective or constraint function files, enter

```
pctRunOnAll('addpath network_file_path')
```

Workers access the function files over the network.

c Check whether a file is on the path of every worker by entering

```
pctRunOnAll('which filename')
```
If any worker does not have a path to the file, it reports

```
filename not found.
```

2 Set your solver to use parallel processing.

MultiStart	patternsearch	ga	particleswarm

`ms =`
`MultiStart('UsePara`
`true);`
 	
or
 	
`ms.UseParallel =`
`true`
	`options =`
	`psoptimset('UsePara`
	`true,`
	`'CompletePoll',`
	`'on',`
	`'Vectorized',`
	`'off');`
		`options =`
		`gaoptimset('UsePara`
		`true,`
		`'Vectorized',`
		`'off');`
			`options =`
			`optimoptions('particleswarm'`
			`'UseParallel',`
			`true,`
			`'Vectorized',`
			`'off');`

	For Optimization app: • **Options > User function evaluation > Evaluate**	For Optimization app: • **Options > User function evaluation > Evaluate fitness**	

MultiStart	patternsearch	ga	particleswarm
	objective and constraint functions > in parallel • **Options > Complete poll > on**	**and constraint functions > in parallel**	

After you establish your parallel computing environment, applicable solvers automatically use parallel computing whenever you call them with `options`.

To stop computing optimizations in parallel, set `UseParallel` to `false`, or set the Optimization app not to compute in parallel. To halt all parallel computation, enter

```
delete(gcp)
```

Parallel Search Functions or Hybrid Functions

To have a `patternsearch` search function run in parallel, or a hybrid function for `ga` or `simulannealbnd` run in parallel, do the following.

1. Set up parallel processing as described in “Multicore Processors” on page 9-12 or “Processor Network” on page 9-13.
2. Ensure that your search function or hybrid function has the conditions outlined in these sections:

- “patternsearch Search Function” on page 9-15
- “Parallel Hybrid Functions” on page 9-17

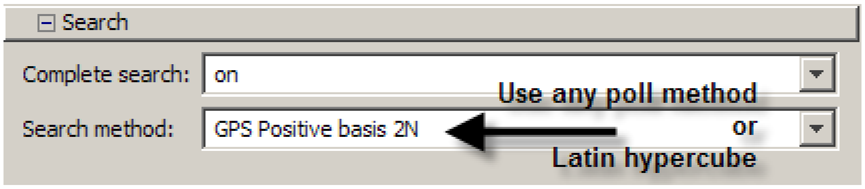
patternsearch Search Function

`patternsearch` uses a parallel search function under the following conditions:

- `CompleteSearch` is `'on'`.
- The search method is not `@searchneldermead` or `custom`.
- If the search method is a `patternsearch` poll method or Latin hypercube search, `UseParallel` is `true`. Set at the command line with `psoptimset`:

```
options = psoptimset('UseParallel',true,...
    'CompleteSearch','on','SearchMethod',@GPSPositiveBasis2N);
```

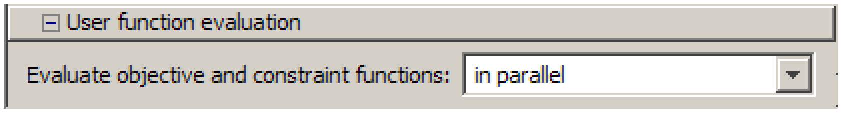
Or you can use the Optimization app.



⊟ Search

Complete search: on

Search method: GPS Positive basis 2N



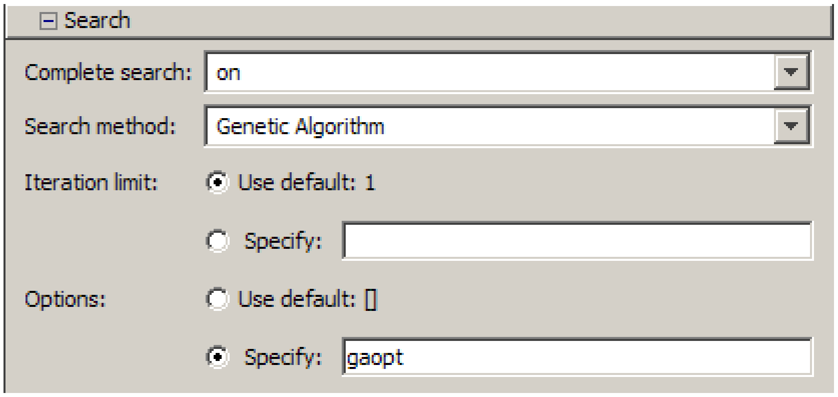
⊟ User function evaluation

Evaluate objective and constraint functions: in parallel

- If the search method is ga, the search method option structure has UseParallel set to true. Set at the command line with psoptimset and gaoptimset:

```
iterlim = 1; % iteration limit, specifies # ga runs
gaopt = gaoptimset('UseParallel',true);
options = psoptimset('SearchMethod',...
    {@searchga,iterlim,gaopt});
```

In the Optimization app, first create the gaopt structure as above, and then use these settings:



⊟ Search

Complete search: on

Search method: Genetic Algorithm

Iteration limit: [✓ Use default: 1] Specify:

Options: Use default: [] [✓ Specify: gaopt]

For more information about search functions, see "Using a Search Method" on page 4-32.

Parallel Hybrid Functions

`ga`, `particleswarm`, and `simulannealbnd` can have other solvers run after or interspersed with their iterations. These other solvers are called hybrid functions. For information on using a hybrid function with `gamultiobj`, see “Parallel Computing with gamultiobj” on page 9-9. Both `patternsearch` and `fmincon` can be hybrid functions. You can set options so that `patternsearch` runs in parallel, or `fmincon` estimates gradients in parallel.

Set the options for the hybrid function as described in “Hybrid Function Options” on page 10-48 for `ga`, “Hybrid Function” on page 10-57 for `particleswarm`, or “Hybrid Function Options” on page 10-66 for `simulannealbnd`. To summarize:

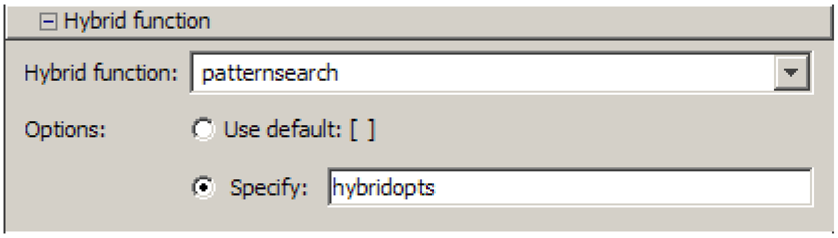
- If your hybrid function is `patternsearch`

  1. Create a `patternsearch` options structure:

     ```
     hybridopts = psoptimset('UseParallel',true,...
         'CompletePoll','on');
     ```
  2. Set the `ga` or `simulannealbnd` options to use `patternsearch` as a hybrid function:

     ```
     options = gaoptimset('UseParallel',true); % for ga
     options = gaoptimset(options,...
         'HybridFcn',{@patternsearch,hybridopts});
     % or, for simulannealbnd:
     options = saoptimset('HybridFcn',{@patternsearch,hybridopts});
     ```

     Or use the Optimization app.

     ⊟ Hybrid function

     Hybrid function: patternsearch

     Options: ○ Use default: [ ]

     ● Specify: hybridopts

  For more information on parallel `patternsearch`, see “Pattern Search” on page 9-6.
- If your hybrid function is `fmincon`:

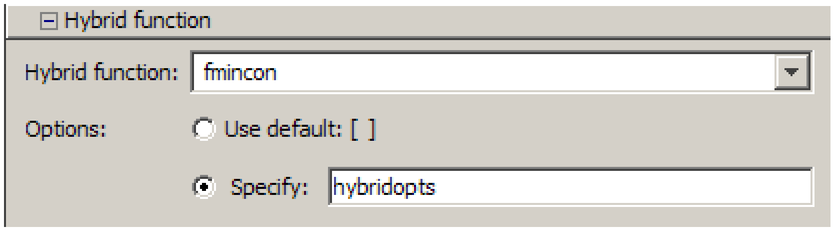
  1. Create a `fmincon` options structure:

```
hybridopts = optimoptions(@fmincon,'UseParallel',true,...
    'Algorithm','interior-point');
% You can use any Algorithm except trust-region-reflective
```

2 Set the `ga` or `simulannealbnd` options to use `fmincon` as a hybrid function:

```
options = gaoptimset('UseParallel',true);
options = gaoptimset(options,'HybridFcn',{@fmincon,hybridopts});
% or, for simulannealbnd:
options = saoptimset('HybridFcn',{@fmincon,hybridopts});
```

Or use the Optimization app.


Hybrid function

Hybrid function: fmincon

Options: Use default: [ ] [✓ Specify: hybridopts]

For more information on parallel `fmincon`, see “Parallel Computing”.

Minimizing an Expensive Optimization Problem Using Parallel Computing Toolbox™

This example shows how to how to speed up the minimization of an expensive optimization problem using functions in Optimization Toolbox™ and Global Optimization Toolbox. In the first part of the example we solve the optimization problem by evaluating functions in a serial fashion and in the second part of the example we solve the same problem using the parallel for loop (`parfor`) feature by evaluating functions in parallel. We compare the time taken by the optimization function in both cases.

Expensive Optimization Problem

For the purpose of this example, we solve a problem in four variables, where the objective and constraint functions are made artificially expensive by pausing.

```
type expensive_objfun.m
type expensive_confun.m
```

```
function f = expensive_objfun(x)
%EXPENSIVE_OBJFUN An expensive objective function used in optimparfor example.

%   Copyright 2007-2014 The MathWorks, Inc.
%   $Revision: 1.1.8.2 $  $Date: 2013/05/04 00:47:14 $

% Simulate an expensive function by pausing
pause(0.1)
% Evaluate objective function
f = exp(x(1)) * (4*x(3)^2 + 2*x(4)^2 + 4*x(1)*x(2) + 2*x(2) + 1);

function [c,ceq] = expensive_confun(x)
%EXPENSIVE_CONFUN An expensive constraint function used in optimparfor example.

%   Copyright 2007-2014 The MathWorks, Inc.
%   $Revision: 1.1.8.2 $  $Date: 2013/05/04 00:47:13 $

% Simulate an expensive function by pausing
pause(0.1);
% Evaluate constraints
c = [1.5 + x(1)*x(2)*x(3) - x(1) - x(2) - x(4);
     -x(1)*x(2) + x(4) - 10];
% No nonlinear equality constraints:
ceq = [];
```

Minimizing Using `fmincon`

We are interested in measuring the time taken by `fmincon` in serial so that we can compare it to the parallel `fmincon` evaluation.

```
startPoint = [-1 1 1 -1];
options = optimoptions('fmincon','Display','iter','Algorithm','sqp');
startTime = tic;
xsol = fmincon(@expensive_objfun,startPoint,[],[],[],[],[],[],@expensive_confun,options
time_fmincon_sequential = toc(startTime);
fprintf('Serial FMINCON optimization takes %g seconds.\n',time_fmincon_sequential);
```

```
                                                          Norm of First-order
 Iter F-count            f(x) Feasibility  Steplength        step  optimality
    0       5    1.839397e+00   1.500e+00                           3.311e+00
    1      12   -8.841073e-01   4.019e+00   4.900e-01   2.335e+00   7.015e-01
    2      17   -1.382832e+00   0.000e+00   1.000e+00   1.142e+00   9.272e-01
    3      22   -2.241952e+00   0.000e+00   1.000e+00   2.447e+00   1.481e+00
    4      27   -3.145762e+00   0.000e+00   1.000e+00   1.756e+00   5.464e+00
    5      32   -5.277523e+00   6.413e+00   1.000e+00   2.224e+00   1.357e+00
    6      37   -6.310709e+00   0.000e+00   1.000e+00   1.099e+00   1.309e+00
    7      43   -6.447956e+00   0.000e+00   7.000e-01   2.191e+00   3.631e+00
    8      48   -7.135133e+00   0.000e+00   1.000e+00   3.719e-01   1.205e-01
    9      53   -7.162732e+00   0.000e+00   1.000e+00   4.083e-01   2.935e-01
   10      58   -7.178390e+00   0.000e+00   1.000e+00   1.591e-01   3.110e-02
   11      63   -7.180399e+00   1.191e-05   1.000e+00   2.644e-02   1.553e-02
   12      68   -7.180408e+00   0.000e+00   1.000e+00   1.140e-02   5.584e-03
   13      73   -7.180411e+00   0.000e+00   1.000e+00   1.764e-03   4.677e-04
   14      78   -7.180412e+00   0.000e+00   1.000e+00   8.827e-05   1.304e-05
   15      83   -7.180412e+00   0.000e+00   1.000e+00   1.528e-06   1.023e-07

Local minimum found that satisfies the constraints.

Optimization completed because the objective function is non-decreasing in
feasible directions, to within the default value of the function tolerance,
and constraints are satisfied to within the default value of the constraint tolerance.



Serial FMINCON optimization takes 18.1397 seconds.
```

Minimizing Using Genetic Algorithm

Since `ga` usually takes many more function evaluations than `fmincon`, we remove the expensive constraint from this problem and perform unconstrained optimization instead; we pass empty ([]) for constraints. In addition, we limit the maximum number of generations to 15 for `ga` so that `ga` can terminate in a reasonable amount of time. We are interested in measuring the time taken by `ga` so that we can compare it to the parallel `ga` evaluation. Note that running `ga` requires Global Optimization Toolbox.

```
rng default % for reproducibility
try
    gaAvailable = false;
    nvar = 4;
    gaoptions = gaoptimset('Generations',15,'Display','iter');
    startTime = tic;
    gasol = ga(@expensive_objfun,nvar,[],[],[],[],[],[],[],gaoptions);
    time_ga_sequential = toc(startTime);
    fprintf('Serial GA optimization takes %g seconds.\n',time_ga_sequential);
    gaAvailable = true;
catch ME
    warning(message('optimdemos:optimparfor:gaNotFound'));
end
```

```
                               Best           Mean      Stall
Generation      f-count        f(x)           f(x)    Generations
    1           100      -6.433e+16      -1.287e+15        0
    2           150      -1.501e+17      -7.138e+15        0
    3           200      -7.878e+26      -1.576e+25        0
    4           250      -8.664e+27      -1.466e+26        0
    5           300      -1.096e+28      -2.062e+26        0
    6           350      -5.422e+33      -1.145e+32        0
    7           400      -1.636e+36      -3.316e+34        0
    8           450      -2.933e+36      -1.513e+35        0
    9           500      -1.351e+40      -2.705e+38        0
   10           550      -1.351e+40        -7.9e+38        1
   11           600       -2.07e+40      -2.266e+39        0
   12           650      -1.845e+44      -3.696e+42        0
   13           700      -2.893e+44      -1.687e+43        0
   14           750      -5.076e+44      -6.516e+43        0
   15           800      -8.321e+44      -2.225e+44        0
Optimization terminated: maximum number of generations exceeded.
Serial GA optimization takes 87.3686 seconds.
```

Setting Parallel Computing Toolbox

The finite differencing used by the functions in Optimization Toolbox to approximate derivatives is done in parallel using the `parfor` feature if Parallel Computing Toolbox is available and there is a parallel pool of workers. Similarly, `ga`, `gamultiobj`, and `patternsearch` solvers in Global Optimization Toolbox evaluate functions in parallel. To use the `parfor` feature, we use the `parpool` function to set up the parallel environment. The computer on which this example is published has four cores, so `parpool` starts four MATLAB® workers. If there is already a parallel pool when you run this example, we use that pool; see the documentation for `parpool` for more information.

```
if max(size(gcp)) == 0 % parallel pool needed
    parpool % create the parallel pool
end
```

```
Starting parallel pool (parpool) using the 'local' profile ... connected to 4 workers.
```

Minimizing Using Parallel `fmincon`

To minimize our expensive optimization problem using the parallel `fmincon` function, we need to explicitly indicate that our objective and constraint functions can be evaluated in parallel and that we want `fmincon` to use its parallel functionality wherever possible. Currently, finite differencing can be done in parallel. We are interested in measuring the time taken by `fmincon` so that we can compare it to the serial `fmincon` run.

```
options = optimoptions(options,'UseParallel',true);
startTime = tic;
xsol = fmincon(@expensive_objfun,startPoint,[],[],[],[],[],[],@expensive_confun,options
time_fmincon_parallel = toc(startTime);
fprintf('Parallel FMINCON optimization takes %g seconds.\n',time_fmincon_parallel);
```

```
                                                          Norm of First-order
 Iter F-count            f(x) Feasibility  Steplength        step  optimality
    0       5    1.839397e+00   1.500e+00                              3.311e+00
    1      12   -8.841073e-01   4.019e+00   4.900e-01   2.335e+00   7.015e-01
    2      17   -1.382832e+00   0.000e+00   1.000e+00   1.142e+00   9.272e-01
    3      22   -2.241952e+00   0.000e+00   1.000e+00   2.447e+00   1.481e+00
    4      27   -3.145762e+00   0.000e+00   1.000e+00   1.756e+00   5.464e+00
    5      32   -5.277523e+00   6.413e+00   1.000e+00   2.224e+00   1.357e+00
    6      37   -6.310709e+00   0.000e+00   1.000e+00   1.099e+00   1.309e+00
    7      43   -6.447956e+00   0.000e+00   7.000e-01   2.191e+00   3.631e+00
    8      48   -7.135133e+00   0.000e+00   1.000e+00   3.719e-01   1.205e-01
    9      53   -7.162732e+00   0.000e+00   1.000e+00   4.083e-01   2.935e-01
   10      58   -7.178390e+00   0.000e+00   1.000e+00   1.591e-01   3.110e-02
   11      63   -7.180399e+00   1.191e-05   1.000e+00   2.644e-02   1.553e-02
   12      68   -7.180408e+00   0.000e+00   1.000e+00   1.140e-02   5.584e-03
   13      73   -7.180411e+00   0.000e+00   1.000e+00   1.764e-03   4.677e-04
   14      78   -7.180412e+00   0.000e+00   1.000e+00   8.827e-05   1.304e-05
   15      83   -7.180412e+00   0.000e+00   1.000e+00   1.528e-06   1.023e-07
```

```
Local minimum found that satisfies the constraints.

Optimization completed because the objective function is non-decreasing in
feasible directions, to within the default value of the function tolerance,
and constraints are satisfied to within the default value of the constraint tolerance.



Parallel FMINCON optimization takes 8.78988 seconds.
```

Minimizing Using Parallel Genetic Algorithm

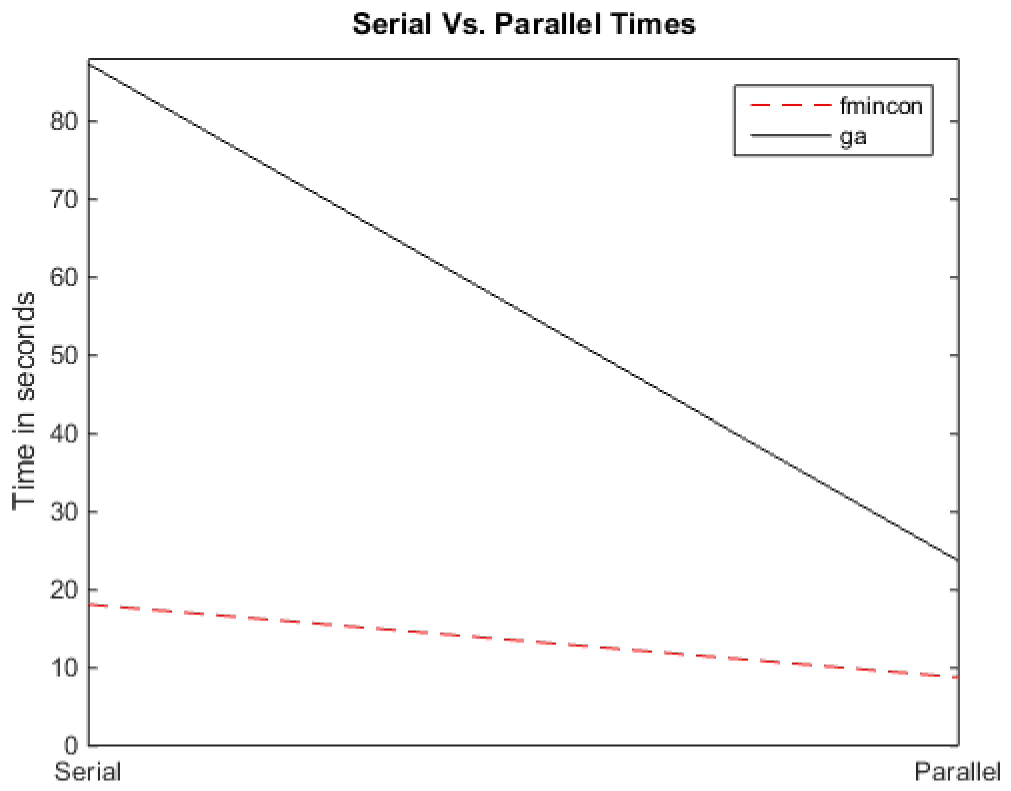
To minimize our expensive optimization problem using the `ga` function, we need to explicitly indicate that our objective function can be evaluated in parallel and that we want `ga` to use its parallel functionality wherever possible. To use the parallel `ga` we also require that the 'Vectorized' option be set to the default (i.e., 'off'). We are again interested in measuring the time taken by `ga` so that we can compare it to the serial `ga` run. Though this run may be different from the serial one because `ga` uses a random number generator, the number of expensive function evaluations is the same in both runs. Note that running `ga` requires Global Optimization Toolbox.

```
rng default % to get the same evaluations as the previous run
if gaAvailable
    gaoptions = gaoptimset(gaoptions,'UseParallel',true);
    startTime = tic;
    gasol = ga(@expensive_objfun,nvar,[],[],[],[],[],[],[],gaoptions);
    time_ga_parallel = toc(startTime);
    fprintf('Parallel GA optimization takes %g seconds.\n',time_ga_parallel);
end
```

```
                               Best           Mean      Stall
Generation      f-count        f(x)           f(x)    Generations
    1           100       -6.433e+16     -1.287e+15        0
    2           150       -1.501e+17     -7.138e+15        0
    3           200       -7.878e+26     -1.576e+25        0
    4           250       -8.664e+27     -1.466e+26        0
    5           300       -1.096e+28     -2.062e+26        0
    6           350       -5.422e+33     -1.145e+32        0
    7           400       -1.636e+36     -3.316e+34        0
    8           450       -2.933e+36     -1.513e+35        0
    9           500       -1.351e+40     -2.705e+38        0
   10           550       -1.351e+40       -7.9e+38        1
   11           600       -2.07e+40      -2.266e+39        0
   12           650      -1.845e+44      -3.696e+42        0
   13           700      -2.893e+44      -1.687e+43        0
   14           750      -5.076e+44      -6.516e+43        0
   15           800      -8.321e+44      -2.225e+44        0
Optimization terminated: maximum number of generations exceeded.
Parallel GA optimization takes 23.707 seconds.
```

Compare Serial and Parallel Time

```
X = [time_fmincon_sequential time_fmincon_parallel];
Y = [time_ga_sequential time_ga_parallel];
t = [0 1];
plot(t,X,'r--',t,Y,'k-')
ylabel('Time in seconds')
legend('fmincon','ga')
ax = gca;
ax.XTick = [0 1];
ax.XTickLabel = {'Serial' 'Parallel'};
axis([0 1 0 ceil(max([X Y]))])
title('Serial Vs. Parallel Times')
```



Utilizing parallel function evaluation via `parfor` improved the efficiency of both `fmincon` and `ga`. The improvement is typically better for expensive objective and constraint functions.

At last we delete the parallel pool.

```
if max(size(gcp)) > 0 % parallel pool exists
    delete(gcp) % delete the pool
end
```

```
Parallel pool using the 'local' profile is shutting down.
```

Options Reference

GlobalSearch and MultiStart Properties (Options)

In this section...
"How to Set Properties" on page 10-2
"Properties of Both Objects" on page 10-2
"GlobalSearch Properties" on page 10-6
"MultiStart Properties" on page 10-7

How to Set Properties

To create a `GlobalSearch` or `MultiStart` object with nondefault properties, use name-value pairs. For example, to create a `GlobalSearch` object that has iterative display and runs only from feasible points with respect to bounds and inequalities, enter

```
gs = GlobalSearch('Display','iter', ...
    'StartPointsToRun','bounds-ineqs');
```

To set a property of an existing `GlobalSearch` or `MultiStart` object, use dot notation. For example, if `ms` is a `MultiStart` object, and you want to set the `Display` property to `'iter'`, enter

```
ms.Display = 'iter';
```

To set multiple properties of an existing object simultaneously, use the constructor (`GlobalSearch` or `MultiStart`) with name-value pairs. For example, to set the `Display` property to `'iter'` and the `MaxTime` property to `100`, enter

```
ms = MultiStart(ms,'Display','iter','MaxTime',100);
```

For more information on setting properties, see "Changing Global Options" on page 3-67.

Properties of Both Objects

You can create a `MultiStart` object from a `GlobalSearch` object and vice-versa.

The syntax for creating a new object from an existing object is:

```
ms =  MultiStart(gs);
```

```
or
gs = GlobalSearch(ms);
```

The new object contains the properties that apply of the old object. This section describes those shared properties:

- “Display” on page 10-3
- “MaxTime” on page 10-3
- “OutputFcns” on page 10-3
- “PlotFcns” on page 10-5
- “StartPointsToRun” on page 10-5
- “TolX” on page 10-5
- “TolFun” on page 10-6

Display

Values for the `Display` property are:

- `'final'` (default) — Summary results to command line after last solver run.
- `'off'` — No output to command line.
- `'iter'` — Summary results to command line after each local solver run.

MaxTime

The `MaxTime` property describes a tolerance on the number of seconds since the solver began its run. Solvers halt when they see `MaxTime` seconds have passed since the beginning of the run. Time means *wall clock* as opposed to processor cycles. The default is `Inf`.

OutputFcns

The `OutputFcns` property directs the global solver to run one or more output functions after each local solver run completes. The output functions also run when the global solver starts and ends. Include a handle to an output function written in the appropriate syntax, or include a cell array of such handles. The default is empty (`[]`).

The syntax of an output function is:

```
stop = outFcn(optimValues,state)
```

- `stop` is a Boolean. When `true`, the algorithm stops. When `false`, the algorithm continues.

  Note: A local solver can have an output function. The global solver does not necessarily stop when a local solver output function causes a local solver run to stop. If you want the global solver to stop in this case, have the global solver output function stop when `optimValues.localsolution.exitflag=-1`.

- `optimValues` is a structure, described in "optimvalues Structure" on page 10-4.
- `state` is a string that indicates the current state of the global algorithm:
  - `'init'` — The global solver has not called the local solver. The fields in the `optimValues` structure are empty, except for `localrunindex`, which is `0`, and `funccount`, which contains the number of objective and constraint function evaluations.
  - `'iter'` — The global solver calls output functions after each local solver run.
  - `'done'` — The global solver finished calling local solvers. The fields in `optimValues` generally have the same values as the ones from the final output function call with `state='iter'`. However, the value of `optimValues.funccount` for `GlobalSearch` can be larger than the value in the last function call with `'iter'`, because the `GlobalSearch` algorithm might have performed some function evaluations that were not part of a local solver. For more information, see "GlobalSearch Algorithm" on page 3-46.

For an example using an output function, see "GlobalSearch Output Function" on page 3-37.

Note: Output and plot functions do not run when `MultiStart` has the `UseParallel` option set to `true` and there is an open `parpool`.

---

optimvalues Structure

The `optimValues` structure contains the following fields:

- `bestx` — The current best point
- `bestfval` — Objective function value at `bestx`
- `funccount` — Total number of function evaluations

- `localrunindex` — Index of the local solver run
- `localsolution` — A structure containing part of the output of the local solver call: `X`, `Fval` and `Exitflag`

PlotFcns

The `PlotFcns` property directs the global solver to run one or more plot functions after each local solver run completes. Include a handle to a plot function written in the appropriate syntax, or include a cell array of such handles. The default is empty (`[]`).

The syntax of a plot function is the same as that of an output function. For details, see "OutputFcns" on page 10-3.

There are two predefined plot functions for the global solvers:

- `@gsplotbestf` plots the best objective function value.
- `@gsplotfunccount` plots the number of function evaluations.

For an example using a plot function, see "MultiStart Plot Function" on page 3-41.

If you specify more than one plot function, all plots appear as subplots in the same window. Right-click any subplot to obtain a larger version in a separate figure window.

Note: Output and plot functions do not run when `MultiStart` has the `UseParallel` option set to `true` and there is an open `parpool`.

StartPointsToRun

The `StartPointsToRun` property directs the solver to exclude certain start points from being run:

- `all` — Accept all start points.
- `bounds` — Reject start points that do not satisfy bounds.
- `bounds-ineqs` — Reject start points that do not satisfy bounds or inequality constraints.

TolX

The `TolX` property describes how close two points must be for solvers to consider them identical for creating the vector of local solutions. Set `TolX` to `0` to obtain the results of every local solver run. Set `TolX` to a larger value to have fewer results. Solvers compute the distance between a pair of points with `norm`, the Euclidean distance.

Solvers consider two solutions identical if they are within `TolX` distance of each other and have objective function values within `TolFun` of each other. If both conditions are not met, solvers report the solutions as distinct.

TolFun

The `TolFun` property describes how close two objective function values must be for solvers to consider them identical for creating the vector of local solutions. Set `TolFun` to `0` to obtain the results of every local solver run. Set `TolFun` to a larger value to have fewer results.

Solvers consider two solutions identical if they are within `TolX` distance of each other and have objective function values within `TolFun` of each other. If both conditions are not met, solvers report the solutions as distinct.

GlobalSearch Properties

NumTrialPoints

Number of potential start points to examine in addition to `x0` from the `problem` structure. `GlobalSearch` runs only those potential start points that pass several tests. For more information, see “GlobalSearch Algorithm” on page 3-46.

Default: `1000`

NumStageOnePoints

Number of start points in Stage 1. For details, see “Obtain Stage 1 Start Point, Run” on page 3-47.

Default: `200`

MaxWaitCycle

A positive integer tolerance appearing in several points in the algorithm.

- If the observed penalty function of `MaxWaitCycle` consecutive trial points is at least the penalty threshold, then raise the penalty threshold (see “PenaltyThresholdFactor” on page 10-7).
- If `MaxWaitCycle` consecutive trial points are in a basin, then update that basin's radius (see “BasinRadiusFactor” on page 10-7).

Default: `20`

BasinRadiusFactor

A basin radius decreases after `MaxWaitCycle` consecutive start points are within the basin. The basin radius decreases by a factor of 1–`BasinRadiusFactor`.

Default: `0.2`

DistanceThresholdFactor

A multiplier for determining whether a trial point is in an existing basin of attraction. For details, see “Examine Stage 2 Trial Point to See if fmincon Runs” on page 3-48. Default: `0.75`

PenaltyThresholdFactor

Determines increase in penalty threshold. For details, see React to Large Counter Values.

Default: `0.2`

MultiStart Properties

UseParallel

The `UseParallel` property determines whether the solver distributes start points to multiple processors:

- `false` (default) — Do not run in parallel.

- `true` — Run in parallel.

For the solver to run in parallel you must set up a parallel environment with `parpool`. For details, see “How to Use Parallel Processing” on page 9-12.

Pattern Search Options

In this section...

Optimization App vs. Command Line

There are two ways to specify options for pattern search, depending on whether you are using the Optimization app or calling the function `patternsearch` at the command line:

- If you are using the Optimization app, you specify the options by selecting an option from a drop-down list or by entering the value of the option in the text field.
- If you are calling `patternsearch` from the command line, you specify the options by creating an `options` structure using the function `psoptimset`, as follows:

  ```
  options = psoptimset('Param1',value1,'Param2',value2,...);
  ```

  See "Set Options" on page 4-44 for examples.

In this section, each option is listed in two ways:

- By its label, as it appears in the Optimization app
- By its field name in the `options` structure

For example:

- **Poll method** refers to the label of the option in the Optimization app.
- `PollMethod` refers to the corresponding field of the `options` structure.

Plot Options

Plot options enable you to plot data from the pattern search while it is running. When you select plot functions and run the pattern search, a plot window displays the plots on separate axes. You can stop the algorithm at any time by clicking the **Stop** button on the plot window.

Plot interval (`PlotInterval`) specifies the number of iterations between consecutive calls to the plot function.

You can select any of the following plots in the **Plot functions** pane.

- **Best function value** (`@psplotbestf`) plots the best objective function value.
- **Function count** (`@psplotfuncount`) plots the number of function evaluations.
- **Mesh size** (`@psplotmeshsize`) plots the mesh size.
- **Best point** (`@psplotbestx`) plots the current best point.
- **Max constraint** (`@psplotmaxconstr`) plots the maximum nonlinear constraint violation.
- **Custom** enables you to use your own plot function. To specify the plot function using the Optimization app,

  - Select **Custom function**.
  - Enter `@myfun` in the text box, where `myfun` is the name of your function.

  “Structure of the Plot Functions” on page 10-11 describes the structure of a plot function.

To display a plot when calling `patternsearch` from the command line, set the `PlotFcns` field of `options` to be a function handle to the plot function. For example, to display the best function value, set `options` as follows

```
options = psoptimset('PlotFcns', @psplotbestf);
```

To display multiple plots, use the syntax

```
options = psoptimset('PlotFcns', {@plotfun1, @plotfun2, ...});
```

where `@plotfun1`, `@plotfun2`, and so on are function handles to the plot functions (listed in parentheses in the preceding list).

If you specify more than one plot function, all plots appear as subplots in the same window. Right-click any subplot to obtain a larger version in a separate figure window.

Structure of the Plot Functions

The first line of a plot function has the form

```
function stop = plotfun(optimvalues, flag)
```

The input arguments to the function are

- `optimvalues` — Structure containing information about the current state of the solver. The structure contains the following fields:
  - `x` — Current point
  - `iteration` — Iteration number
  - `fval` — Objective function value
  - `meshsize` — Current mesh size
  - `funccount` — Number of function evaluations
  - `method` — Method used in last iteration
  - `TolFun` — Tolerance on function value in last iteration
  - `TolX` — Tolerance on `x` value in last iteration
  - `nonlinineq` — Nonlinear inequality constraints, displayed only when a nonlinear constraint function is specified
  - `nonlineq` — Nonlinear equality constraints, displayed only when a nonlinear constraint function is specified
- `flag` — Current state in which the plot function is called. The possible values for `flag` are
  - `init` — Initialization state
  - `iter` — Iteration state
  - `interrupt` — Intermediate stage
  - `done` — Final state

"Passing Extra Parameters" in the Optimization Toolbox documentation explains how to provide additional parameters to the function.

The output argument `stop` provides a way to stop the algorithm at the current iteration. `stop` can have the following values:

- `false` — The algorithm continues to the next iteration.
- `true` — The algorithm terminates at the current iteration.

Poll Options

Poll options control how the pattern search polls the mesh points at each iteration.

Poll method (`PollMethod`) specifies the pattern the algorithm uses to create the mesh. There are two patterns for each of the classes of direct search algorithms: the generalized pattern search (GPS) algorithm, the generating set search (GSS) algorithm, and the mesh adaptive direct search (MADS) algorithm. These patterns are the Positive basis 2*N* and the Positive basis *N*+1:

- The default pattern, `GPS Positive basis 2N` (`GPSPositiveBasis2N`), consists of the following 2*N* vectors, where *N* is the number of independent variables for the objective function.
  [1 0 0...0][0 1 0...0] ...[0 0 0...1][–1 0 0...0][0 –1 0...0][0 0 0...–1].

  For example, if the optimization problem has three independent variables, the pattern consists of the following six vectors.
  [1 0 0][0 1 0][0 0 1][–1 0 0][0 –1 0][0 0 –1].
- The `GSS Positive basis 2N` pattern (`GSSPositiveBasis2N`) is similar to `GPS Positive basis 2N`, but adjusts the basis vectors to account for linear constraints. `GSS Positive basis 2N` is more efficient than `GPS Positive basis 2N` when the current point is near a linear constraint boundary.
- The `MADS Positive basis 2N` pattern (`MADSPositiveBasis2N`) consists of 2*N* randomly generated vectors, where *N* is the number of independent variables for the objective function. This is done by randomly generating *N* vectors which form a linearly independent set, then using this first set and the negative of this set gives 2*N* vectors. As shown above, the `GPS Positive basis 2N` pattern is formed using the positive and negative of the linearly independent identity, however, with the `MADS Positive basis 2N`, the pattern is generated using a random permutation of an *N*-by-*N* linearly independent lower triangular matrix that is regenerated at each iteration.

- The `GPS Positive basis NP1` pattern consists of the following $N + 1$ vectors.
  [1 0 0...0][0 1 0...0] ...[0 0 0...1][–1 –1 –1...–1].

  For example, if the objective function has three independent variables, the pattern consists of the following four vectors.
  [1 0 0][0 1 0][0 0 1][–1 –1 –1].
- The `GSS Positive basis Np1` pattern (`GSSPositiveBasisNp1`) is similar to `GPS Positive basis Np1`, but adjusts the basis vectors to account for linear constraints. `GSS Positive basis Np1` is more efficient than `GPS Positive basis Np1` when the current point is near a linear constraint boundary.
- The `MADS Positive basis Np1` pattern (`MADSPositiveBasisNp1`) consists of N randomly generated vectors to form the positive basis, where N is the number of independent variables for the objective function. Then, one more random vector is generated, giving N+1 randomly generated vectors. Each iteration generates a new pattern when the `MADS Positive basis N+1` is selected.

Complete poll (`CompletePoll`) specifies whether all the points in the current mesh must be polled at each iteration. **Complete Poll** can have the values `On` or `Off`.

- If you set **Complete poll** to `On`, the algorithm polls all the points in the mesh at each iteration and chooses the point with the smallest objective function value as the current point at the next iteration.
- If you set **Complete poll** to `Off`, the default value, the algorithm stops the poll as soon as it finds a point whose objective function value is less than that of the current point. The algorithm then sets that point as the current point at the next iteration.

Polling order (`PollingOrder`) specifies the order in which the algorithm searches the points in the current mesh. The options are

- `Random` — The polling order is random.
- `Success` — The first search direction at each iteration is the direction in which the algorithm found the best point at the previous iteration. After the first point, the algorithm polls the mesh points in the same order as **Consecutive**.
- `Consecutive` — The algorithm polls the mesh points in *consecutive* order, that is, the order of the pattern vectors as described in “Poll Method” on page 4-21.

Search Options

Search options specify an optional search that the algorithm can perform at each iteration prior to the polling. If the search returns a point that improves the objective function, the algorithm uses that point at the next iteration and omits the polling. Please note, if you have selected the same **Search method** and **Poll method**, only the option selected in the Poll method will be used, although both will be used when the options selected are different.

Complete search (`CompleteSearch`) applies when you set **Search method** to `GPS Positive basis Np1`, `GPS Positive basis 2N`, `GSS Positive basis Np1`, `GSS Positive basis 2N`, `MADS Positive basis Np1`, `MADS Positive basis 2N`, or `Latin hypercube`. **Complete search** can have the values `On` or `Off`.

For `GPS Positive basis Np1`, `MADS Positive basis Np1, GPS Positive basis 2N`, or `MADS Positive basis 2N`, **Complete search** has the same meaning as the poll option **Complete poll**.

Search method (`SearchMethod`) specifies the optional search step. The options are

- `None` (`[]`) (the default) specifies no search step.
- `GPS Positive basis 2N` (`@GPSPositiveBasis2N`)
- `GPS Positive basis Np1` (`@GPSPositiveBasisNp1`)
- `GSS Positive basis 2N` (`@GSSPositiveBasis2N`)
- `GSS Positive basis Np1` (`@GSSPositiveBasisNp1`)
- `MADS Positive basis 2N` (`@MADSPositiveBasis2N`)
- `MADS Positive basis Np1` (`@MADSPositiveBasisNp1`)
- `Genetic Algorithm` (`@searchga`) specifies a search using the genetic algorithm. If you select `Genetic Algorithm`, two other options appear:
  - **Iteration limit** — Positive integer specifying the number of iterations of the pattern search for which the genetic algorithm search is performed. The default for **Iteration limit** is 1.
  - **Options** — Options structure for the genetic algorithm, which you can set using `gaoptimset`.

  To change the default values of **Iteration limit** and **Options** at the command line, use the syntax

  ```
  options = psoptimset('SearchMethod',...
        {@searchga,iterlim,optionsGA})
  ```

  where `iterlim` is the value of **Iteration limit** and `optionsGA` is the genetic algorithm options structure.

Note If you set **Search method** to `Genetic algorithm` or `Nelder-Mead`, we recommend that you leave **Iteration limit** set to the default value `1`, because performing these searches more than once is not likely to improve results.

- `Latin hypercube` (`@searchlhs`) specifies a Latin hypercube search. `patternsearch` generates each point for the search as follows. For each component, take a random permutation of the vector `[1,2,...,k]` minus `rand(1,k)`, divided by `k`. (`k` is the number of points.) This yields `k` points, with each component close to evenly spaced. The resulting points are then scaled to fit any bounds. `Latin hypercube` uses default bounds of `-1` and `1`.

  The way the search is performed depends on the setting for **Complete search**:

  - If you set **Complete search** to `On`, the algorithm polls all the points that are randomly generated at each iteration by the Latin hypercube search and chooses the one with the smallest objective function value.
  - If you set **Complete search** to `Off` (the default), the algorithm stops the poll as soon as it finds one of the randomly generated points whose objective function value is less than that of the current point, and chooses that point for the next iteration.

  If you select `Latin hypercube`, two other options appear:

  - **Iteration limit** — Positive integer specifying the number of iterations of the pattern search for which the Latin hypercube search is performed. The default for **Iteration limit** is 1.
  - **Design level** — The **Design level** is the number of points `patternsearch` searches, a positive integer. The default for **Design level** is 15 times the number of dimensions.

  To change the default values of **Iteration limit** and **Design level** at the command line, use the syntax

  ```
  options=psoptimset('SearchMethod', {@searchlhs,iterlim,level})
  ```

  where `iterlim` is the value of **Iteration limit** and `level` is the value of **Design level**.

- `Nelder-Mead` (`@searchneldermead`) specifies a search using `fminsearch`, which uses the Nelder-Mead algorithm. If you select `Nelder-Mead`, two other options appear:

  - **Iteration limit** — Positive integer specifying the number of iterations of the pattern search for which the Nelder-Mead search is performed. The default for **Iteration limit** is 1.
  - **Options** — Options structure for the function `fminsearch`, which you can create using the function `optimset`.

  To change the default values of **Iteration limit** and **Options** at the command line, use the syntax

  ```
  options=psoptimset('SearchMethod',...
       {@searchneldermead,iterlim,optionsNM})
  ```

  where `iterlim` is the value of **Iteration limit** and `optionsNM` is the options structure.
- `Custom` enables you to write your own search function. To specify the search function using the Optimization app,

  - Set **Search function** to `Custom`.
  - Set **Function name** to `@myfun`, where `myfun` is the name of your function.

  If you are using `patternsearch`, set

  ```
  options = psoptimset('SearchMethod', @myfun);
  ```

  To see a template that you can use to write your own search function, enter

  ```
  edit searchfcntemplate
  ```

  The following section describes the structure of the search function.

Structure of the Search Function

Your search function must have the following calling syntax.

```
function [successSearch,xBest,fBest,funccount] =
searchfcntemplate(fun,x,A,b,Aeq,beq,lb,ub, ...
    optimValues,options)
```

The search function has the following input arguments:

- `fun` — Objective function
- `x` — Current point
- `A,b` — Linear inequality constraints
- `Aeq,beq` — Linear equality constraints
- `lb,ub` — Lower and upper bound constraints
- `optimValues` — Structure that enables you to set search options. The structure contains the following fields:
  - `x` — Current point
  - `fval` — Objective function value at `x`
  - `iteration` — Current iteration number
  - `funccount` — Counter for user function evaluation
  - `scale` — Scale factor used to scale the design points
  - `problemtype` — Flag passed to the search routines, indicating whether the problem is `'unconstrained'`, `'boundconstraints'`, or `'linearconstraints'`. This field is a subproblem type for nonlinear constrained problems.
  - `meshsize` — Current mesh size used in search step
  - `method` — Method used in last iteration
- `options` — Pattern search options structure

The function has the following output arguments:

- `successSearch` — A Boolean identifier indicating whether the search is successful or not
- `xBest,fBest` — Best point and best function value found by search method
- `funccount` — Number of user function evaluation in search method

See "Using a Search Method" on page 4-32 for an example.

Mesh Options

Mesh options control the mesh that the pattern search uses. The following options are available.

Initial size (`InitialMeshSize`) specifies the size of the initial mesh, which is the length of the shortest vector from the initial point to a mesh point. **Initial size** should be a positive scalar. The default is `1.0`.

Max size (`MaxMeshSize`) specifies a maximum size for the mesh. When the maximum size is reached, the mesh size does not increase after a successful iteration. **Max size** must be a positive scalar, and is only used when a GPS or GSS algorithm is selected as the Poll or Search method. The default value is `Inf`. MADS uses a maximum size of `1`.

Accelerator (`MeshAccelerator`) specifies whether, when the mesh size is small, the **Contraction factor** is multiplied by `0.5` after each unsuccessful iteration. **Accelerator** can have the values `On` or `Off`, the default. **Accelerator** applies to the GPS and GSS algorithms.

Rotate (`MeshRotate`) is only applied when **Poll method** is set to `GPS Positive basis Np1` or `GSS Positive basis Np1`. It specifies whether the mesh vectors are multiplied by –1 when the mesh size is less than 1/100 of the mesh tolerance (minimum mesh size `TolMesh`) after an unsuccessful poll. In other words, after the first unsuccessful poll with small mesh size, instead of polling in directions e_i (unit positive directions) and $-\Sigma e_i$, the algorithm polls in directions $-e_i$ and Σe_i. **Rotate** can have the values `Off` or `On` (the default). When the problem has equality constraints, **Rotate** is disabled.

Rotate is especially useful for discontinuous functions.

Note Changing the setting of **Rotate** has no effect on the poll when **Poll method** is set to `GPS Positive basis 2N`, `GSS Positive basis 2N`, `MADS Positive basis 2N`, or `MADS Positive basis Np1`.

Scale (`ScaleMesh`) specifies whether the algorithm scales the mesh points by carefully multiplying the pattern vectors by constants proportional to the logarithms of the absolute values of components of the current point (or, for unconstrained problems, of the initial point). **Scale** can have the values `Off` or `On` (the default). When the problem has equality constraints, **Scale** is disabled.

Expansion factor (`MeshExpansion`) specifies the factor by which the mesh size is increased after a successful poll. The default value is `2.0`, which means that the size of the mesh is multiplied by `2.0` after a successful poll. **Expansion factor** must be a positive scalar and is only used when a GPS or GSS method is selected as the Poll or Search method. MADS uses a factor of `4.0`.

Contraction factor (`MeshContraction`) specifies the factor by which the mesh size is decreased after an unsuccessful poll. The default value is `0.5`, which means that the size of the mesh is multiplied by `0.5` after an unsuccessful poll. **Contraction factor** must be a positive scalar and is only used when a GPS or GSS method is selected as the Poll or Search method. MADS uses a factor of `0.25`.

See "Mesh Expansion and Contraction" on page 4-58 for more information.

Constraint Parameters

For information on the meaning of penalty parameters, see "Nonlinear Constraint Solver Algorithm" on page 4-37.

- **Initial penalty** (`InitialPenalty`) — Specifies an initial value of the penalty parameter that is used by the nonlinear constraint algorithm. **Initial penalty** must be greater than or equal to `1`, and has a default of `10`.
- **Penalty factor** (`PenaltyFactor`) — Increases the penalty parameter when the problem is not solved to required accuracy and constraints are not satisfied. **Penalty factor** must be greater than `1`, and has a default of `100`.

Bind tolerance (`TolBind`) specifies the tolerance for the distance from the current point to the boundary of the feasible region with respect to linear constraints. This means `Bind tolerance` specifies when a linear constraint is active. `Bind tolerance` is not a stopping criterion. Active linear constraints change the pattern of points `patternsearch` uses for polling or searching. `patternsearch` always uses points that satisfy linear constraints to within `Bind tolerance`. The default value of `Bind tolerance` is `1e-3`.

Cache Options

The pattern search algorithm can keep a record of the points it has already polled, so that it does not have to poll the same point more than once. If the objective function requires a relatively long time to compute, the cache option can speed up the algorithm. The memory allocated for recording the points is called the cache. This option should only be used for deterministic objective functions, but not for stochastic ones.

Cache (`Cache`) specifies whether a cache is used. The options are `On` and `Off`, the default. When you set **Cache** to `On`, the algorithm does not evaluate the objective function at any mesh points that are within **Tolerance** of a point in the cache.

Tolerance (`CacheTol`) specifies how close a mesh point must be to a point in the cache for the algorithm to omit polling it. **Tolerance** must be a positive scalar. The default value is `eps`.

Size (`CacheSize`) specifies the size of the cache. **Size** must be a positive scalar. The default value is `1e4`.

See “Use Cache” on page 4-75 for more information.

Stopping Criteria

Stopping criteria determine what causes the pattern search algorithm to stop. Pattern search uses the following criteria:

Mesh tolerance (`TolMesh`) specifies the minimum tolerance for mesh size. The GPS and GSS algorithms stop if the mesh size becomes smaller than **Mesh tolerance**. MADS 2N stops when the mesh size becomes smaller than `TolMesh^2`. MADS Np1 stops when the mesh size becomes smaller than `(TolMesh/nVar)^2`, where `nVar` is the number of elements of `x0`. The default value of `TolMesh` is `1e-6`.

Max iteration (`MaxIter`) specifies the maximum number of iterations the algorithm performs. The algorithm stops if the number of iterations reaches **Max iteration**. You can select either

- **100*numberOfVariables** — Maximum number of iterations is 100 times the number of independent variables (the default).
- **Specify** — A positive integer for the maximum number of iterations

Max function evaluations (`MaxFunEval`) specifies the maximum number of evaluations of the objective function. The algorithm stops if the number of function evaluations reaches **Max function evaluations**. You can select either

- **2000*numberOfVariables** — Maximum number of function evaluations is 2000 times the number of independent variables.
- **Specify** — A positive integer for the maximum number of function evaluations

Time limit (`TimeLimit`) specifies the maximum time in seconds the pattern search algorithm runs before stopping. This also includes any specified pause time for pattern search algorithms.

X tolerance (`TolX`) specifies the minimum distance between the current points at two consecutive iterations. The algorithm stops if the distance between two consecutive points is less than **X tolerance** and the mesh size is smaller than **X tolerance**. The default value is `1e-6`.

Function tolerance (`TolFun`) specifies the minimum tolerance for the objective function. After a successful poll, the algorithm stops if the difference between the function value at the previous best point and function value at the current best point is less than **Function tolerance**, and if the mesh size is also smaller than **Function tolerance**. The default value is `1e-6`.

See “Setting Solver Tolerances” on page 4-31 for an example.

Constraint tolerance (`TolCon`) — The **Constraint tolerance** is not used as stopping criterion. It is used to determine the feasibility with respect to nonlinear constraints.

Output Function Options

Output functions are functions that the pattern search algorithm calls at each generation. To specify the output function using the Optimization app,

- Select **Custom function**.
- Enter `@myfun` in the text box, where `myfun` is the name of your function.
- To pass extra parameters in the output function, use “Anonymous Functions”.
- For multiple output functions, enter a cell array of output function handles: `{@myfun1,@myfun2,...}`.

At the command line, set

```
options = psoptimset('OutputFcns',@myfun);
```

For multiple output functions, enter a cell array:

```
options = psoptimset('OutputFcns',{@myfun1,@myfun2,...});
```

To see a template that you can use to write your own output function, enter

```
edit psoutputfcntemplate
```

at the MATLAB command prompt.

The following section describes the structure of the output function.

Structure of the Output Function

Your output function must have the following calling syntax:

```
[stop,options,optchanged] = myfun(optimvalues,options,flag)
```

The function has the following input arguments:

- `optimvalues` — Structure containing information about the current state of the solver. The structure contains the following fields:
  - `x` — Current point
  - `iteration` — Iteration number
  - `fval` — Objective function value
  - `meshsize` — Current mesh size
  - `funccount` — Number of function evaluations
  - `method` — Method used in last iteration
  - `TolFun` — Tolerance on function value in last iteration
  - `TolX` — Tolerance on `x` value in last iteration
  - `nonlinineq` — Nonlinear inequality constraints, displayed only when a nonlinear constraint function is specified
  - `nonlineq` — Nonlinear equality constraints, displayed only when a nonlinear constraint function is specified
- `options` — Options structure
- `flag` — Current state in which the output function is called. The possible values for `flag` are
  - `init` — Initialization state
  - `iter` — Iteration state
  - `interrupt` — Intermediate stage
  - `done` — Final state

"Passing Extra Parameters" in the Optimization Toolbox documentation explains how to provide additional parameters to the output function.

The output function returns the following arguments to `ga`:

- `stop` — Provides a way to stop the algorithm at the current iteration. `stop` can have the following values.
  - `false` — The algorithm continues to the next iteration.
  - `true` — The algorithm terminates at the current iteration.
- `options` — Options structure.
- `optchanged` — Flag indicating changes to `options`.

Display to Command Window Options

Level of display (`'Display'`) specifies how much information is displayed at the command line while the pattern search is running. The available options are

- `Off` (`'off'`) — No output is displayed.
- `Iterative` (`'iter'`) — Information is displayed for each iteration.
- `Diagnose` (`'diagnose'`) — Information is displayed for each iteration. In addition, the diagnostic lists some problem information and the options that are changed from the defaults.
- `Final` (`'final'`) — The reason for stopping is displayed.

Both `Iterative` and `Diagnose` display the following information:

- `Iter` — Iteration number
- `FunEval` — Cumulative number of function evaluations
- `MeshSize` — Current mesh size
- `FunVal` — Objective function value of the current point
- `Method` — Outcome of the current poll (with no nonlinear constraint function specified). With a nonlinear constraint function, `Method` displays the update method used after a subproblem is solved.
- `Max Constraint` — Maximum nonlinear constraint violation (displayed only when a nonlinear constraint function has been specified)

The default value of **Level of display** is

- `Off` in the Optimization app

- `'final'` in an options structure created using `psoptimset`

Vectorize and Parallel Options (User Function Evaluation)

You can choose to have your objective and constraint functions evaluated in serial, parallel, or in a vectorized fashion. These options are available in the **User function evaluation** section of the **Options** pane of the Optimization app, or by setting the `'Vectorized'` and `'UseParallel'` options with `psoptimset`.

Note: You must set `CompletePoll` to `'on'` for `patternsearch` to use vectorized or parallel polling. Similarly, set `CompleteSearch` to `'on'` for vectorized or parallel searching.

- When **Evaluate objective and constraint functions** (`'Vectorized'`) is **in serial** (`'Off'`), `patternsearch` calls the objective function on one point at a time as it loops through the mesh points. (At the command line, this assumes `'UseParallel'` is at its default value of `false`.)
- When **Evaluate objective and constraint functions** (`'Vectorized'`) is **vectorized** (`'On'`), `patternsearch` calls the objective function on all the points in the mesh at once, i.e., in a single call to the objective function.

  If there are nonlinear constraints, the objective function and the nonlinear constraints all need to be vectorized in order for the algorithm to compute in a vectorized manner.

  For details and an example, see “Vectorize the Objective and Constraint Functions” on page 4-80.
- When **Evaluate objective and constraint functions** (`UseParallel`) is **in parallel** (`true`), `patternsearch` calls the objective function in parallel, using the parallel environment you established (see “How to Use Parallel Processing” on page 9-12). At the command line, set `UseParallel` to `false` to compute serially.

Note: You cannot simultaneously use vectorized and parallel computations. If you set `'UseParallel'` to `true` and `'Vectorized'` to `'On'`, `patternsearch` evaluates your objective and constraint functions in a vectorized manner, not in parallel.

How Objective and Constraint Functions Are Evaluated

Assume `CompletePoll = 'on'`	`Vectorized = 'off'`	`Vectorized = 'on'`
`UseParallel = false`	Serial	Vectorized
`UseParallel = true`	Parallel	Vectorized

Options Table for Pattern Search Algorithms

Option Availability Table for All Algorithms

Option	Description	Algorithm Availability
`Cache`	With `Cache` set to `'on'`, `patternsearch` keeps a history of the mesh points it polls and does not poll points close to them again at subsequent iterations. Use this option if `patternsearch` runs slowly because it is taking a long time to compute the objective function. If the objective function is stochastic, it is advised not to use this option.	All
`CacheSize`	Size of the cache, in number of points.	All
`CacheTol`	Positive scalar specifying how close the current mesh point must be to a point in the cache in order for `patternsearch` to avoid polling it. Available if `'Cache'` option is set to `'on'`.	All
`CompletePoll`	Complete poll around current iterate. Evaluate all the points in a poll step.	All
`CompleteSearch`	Complete search around current iterate when the search method is a poll method. Evaluate all the points in a search step.	All

Option	Description	Algorithm Availability
`Display`	Level of display to Command Window.	All
`InitialMeshSize`	Initial mesh size used in pattern search algorithms.	All
`InitialPenalty`	Initial value of the penalty parameter.	All
`MaxFunEvals`	Maximum number of objective function evaluations.	All
`MaxIter`	Maximum number of iterations.	All
`MaxMeshSize`	Maximum mesh size used in a poll/search step.	GPS and GSS
`MeshAccelerator`	Accelerate mesh size contraction.	GPS and GSS
`MeshContraction`	Mesh contraction factor, used when iteration is unsuccessful.	GPS and GSS
`MeshExpansion`	Mesh expansion factor, expands mesh when iteration is successful.	GPS and GSS
`MeshRotate`	Rotate the pattern before declaring a point to be optimum.	GPS Np1 and GSS Np1
`OutputFcns`	User-specified function that a pattern search calls at each iteration.	All
`PenaltyFactor`	Penalty update parameter.	All
`PlotFcns`	Specifies function to plot at runtime.	All
`PlotInterval`	Specifies that plot functions will be called at every interval.	All
`PollingOrder`	Order in which search directions are polled.	GPS and GSS
`PollMethod`	Polling strategy used in pattern search.	All

Option	Description	Algorithm Availability
`ScaleMesh`	Automatic scaling of variables.	All
`SearchMethod`	Specifies search method used in pattern search.	All
`TimeLimit`	Total time (in seconds) allowed for optimization. Also includes any specified pause time for pattern search algorithms.	All
`TolBind`	Binding tolerance used to determine if linear constraint is active.	All
`TolCon`	Tolerance on nonlinear constraints.	All
`TolFun`	Tolerance on function value.	All
`TolMesh`	Tolerance on mesh size.	All
`TolX`	Tolerance on independent variable.	All
`UseParallel`	When `true`, compute objective functions of a poll or search in parallel. Disable by setting to `false`.	All
`Vectorized`	Specifies whether objective and constraint functions are vectorized.	All

Genetic Algorithm Options

In this section...

Optimization App vs. Command Line

There are two ways to specify options for the genetic algorithm, depending on whether you are using the Optimization app or calling the functions `ga` or `gamultiobj` at the command line:

- If you are using the Optimization app (`optimtool`), select an option from a drop-down list or enter the value of the option in a text field.
- If you are calling `ga` or `gamultiobj` at the command line, create an `options` structure using the function `gaoptimset`, as follows:

  ```
  options = gaoptimset('Param1', value1, 'Param2', value2, ...);
  ```

  See "Setting Options at the Command Line" on page 5-64 for examples.

In this section, each option is listed in two ways:

- By its label, as it appears in the Optimization app
- By its field name in the `options` structure

For example:

- **Population type** is the label of the option in the Optimization app.
- `PopulationType` is the corresponding field of the `options` structure.

Plot Options

Plot options let you plot data from the genetic algorithm while it is running. You can stop the algorithm at any time by clicking the **Stop** button on the plot window.

Plot interval (`PlotInterval`) specifies the number of generations between consecutive calls to the plot function.

You can select any of the following plot functions in the **Plot functions** pane:

- **Best fitness** (`@gaplotbestf`) plots the best function value versus generation.
- **Expectation** (`@gaplotexpectation`) plots the expected number of children versus the raw scores at each generation.
- **Score diversity** (`@gaplotscorediversity`) plots a histogram of the scores at each generation.
- **Stopping** (`@gaplotstopping`) plots stopping criteria levels.
- **Best individual** (`@gaplotbestindiv`) plots the vector entries of the individual with the best fitness function value in each generation.
- **Genealogy** (`@gaplotgenealogy`) plots the genealogy of individuals. Lines from one generation to the next are color-coded as follows:
  - Red lines indicate mutation children.
  - Blue lines indicate crossover children.
  - Black lines indicate elite individuals.
- **Scores** (`@gaplotscores`) plots the scores of the individuals at each generation.
- **Max constraint** (`@gaplotmaxconstr`) plots the maximum nonlinear constraint violation at each generation.

- **Distance** (`@gaplotdistance`) plots the average distance between individuals at each generation.
- **Range** (`@gaplotrange`) plots the minimum, maximum, and mean fitness function values in each generation.
- **Selection** (`@gaplotselection`) plots a histogram of the parents.
- **Custom function** lets you use plot functions of your own. To specify the plot function if you are using the Optimization app,
  - Select **Custom function**.
  - Enter `@myfun` in the text box, where `myfun` is the name of your function.

  See "Structure of the Plot Functions" on page 10-31.

`gamultiobj` allows **Distance**, **Genealogy**, **Score diversity**, **Selection**, **Stopping**, and **Custom function**, as well as the following additional choices:

- **Pareto front** (`@gaplotpareto`) plots the Pareto front for the first two objective functions.
- **Average Pareto distance** (`@gaplotparetodistance`) plots a bar chart of the distance of each individual from its neighbors.
- **Rank histogram** (`@gaplotrankhist`) plots a histogram of the ranks of the individuals. Individuals of rank 1 are on the Pareto frontier. Individuals of rank 2 are lower than at least one rank 1 individual, but are not lower than any individuals from other ranks, etc.
- **Average Pareto spread** (`@gaplotspread`) plots the average spread as a function of iteration number.

To display a plot when calling `ga` from the command line, set the `PlotFcns` field of `options` to be a function handle to the plot function. For example, to display the best fitness plot, set `options` as follows:

```
options = gaoptimset('PlotFcns', @gaplotbestf);
```

To display multiple plots, use the syntax

```
options = gaoptimset('PlotFcns', {@plotfun1, @plotfun2, ...});
```

where `@plotfun1`, `@plotfun2`, and so on are function handles to the plot functions.

If you specify multiple plot functions, all plots appear as subplots in the same window. Right-click any subplot to obtain a larger version in a separate figure window.

Structure of the Plot Functions

The first line of a plot function has this form:

```
function state = plotfun(options,state,flag)
```

The input arguments to the function are

- `options` — Structure containing all the current options settings.
- `state` — Structure containing information about the current generation. “The State Structure” on page 10-31 describes the fields of `state`.
- `flag` — String that tells what stage the algorithm is currently in.

“Passing Extra Parameters” in the Optimization Toolbox documentation explains how to provide additional parameters to the function.

The output argument `state` is a state structure as well. Pass the input argument, modified if you like. To stop the iterations, set `state.StopFlag` to a nonempty string.

The State Structure

ga

The state structure for `ga`, which is an input argument to plot, mutation, and output functions, contains the following fields:

- `Population` — Population in the current generation
- `Score` — Scores of the current population
- `Generation` — Current generation number
- `StartTime` — Time when genetic algorithm started
- `StopFlag` — String containing the reason for stopping
- `Selection` — Indices of individuals selected for elite, crossover, and mutation
- `Expectation` — Expectation for selection of individuals
- `Best` — Vector containing the best score in each generation
- `LastImprovement` — Generation at which the last improvement in fitness value occurred
- `LastImprovementTime` — Time at which last improvement occurred
- `NonlinIneq` — Nonlinear inequality constraints, displayed only when a nonlinear constraint function is specified

- `NonlinEq` — Nonlinear equality constraints, displayed only when a nonlinear constraint function is specified

gamultiobj

The state structure for `gamultiobj`, which is an input argument to plot, mutation, and output functions, contains the following fields:

- `Population` — Population in the current generation
- `Score` — Scores of the current population, a `Population`-by-`nObjectives` matrix, where `nObjectives` is the number of objectives
- `Generation` — Current generation number
- `StartTime` — Time when genetic algorithm started
- `StopFlag` — String containing the reason for stopping
- `Selection` — Indices of individuals selected for elite, crossover, and mutation
- `Rank` — Vector of the ranks of members in the population
- `Distance` — Vector of distances of each member of the population to the nearest neighboring member
- `AverageDistance` — The average of `Distance`
- `Spread` — Vector where the entries are the spread in each generation

Population Options

Population options let you specify the parameters of the population that the genetic algorithm uses.

Population type (`PopulationType`) specifies the type of input to the fitness function. Types and their restrictions are:

- `Double vector` (`'doubleVector'`) — Use this option if the individuals in the population have type `double`. Use this option for mixed integer programming. This is the default.
- `Bit string` (`'bitstring'`) — Use this option if the individuals in the population have components that are `0` or `1`.

Caution The individuals in a `Bit string` population are vectors of type `double`, not strings.

For **Creation function** (`CreationFcn`) and **Mutation function** (`MutationFcn`), use `Uniform` (`@gacreationuniform` and `@mutationuniform`) or `Custom`. For **Crossover function** (`CrossoverFcn`), use `Scattered` (`@crossoverscattered`), `Single point` (`@crossoversinglepoint`), `Two point` (`@crossovertwopoint`), or `Custom`. You cannot use a **Hybrid function**, and `ga` ignores all constraints, including bounds, linear constraints, and nonlinear constraints.

- `Custom` — For **Crossover function** and **Mutation function**, use `Custom`. For **Creation function**, either use `Custom`, or provide an **Initial population**. You cannot use a **Hybrid function**, and `ga` ignores all constraints, including bounds, linear constraints, and nonlinear constraints.

Population size (`PopulationSize`) specifies how many individuals there are in each generation. With a large population size, the genetic algorithm searches the solution space more thoroughly, thereby reducing the chance that the algorithm returns a local minimum that is not a global minimum. However, a large population size also causes the algorithm to run more slowly.

If you set **Population size** to a vector, the genetic algorithm creates multiple subpopulations, the number of which is the length of the vector. The size of each subpopulation is the corresponding entry of the vector. See "Migration Options" on page 10-46.

Creation function (`CreationFcn`) specifies the function that creates the initial population for `ga`. Do not specify a creation function with integer problems because `ga` overrides any choice you make. Choose from:

- `[]` uses the default creation function for your problem.
- `Uniform` (`@gacreationuniform`) creates a random initial population with a uniform distribution. This is the default when there are no linear constraints, or when there are integer constraints. The uniform distribution is in the initial population range (`PopInitRange`). The default values for `PopInitRange` are `[-10;10]` for every component, or `[-9999;10001]` when there are integer constraints. These bounds are shifted and scaled to match any existing bounds `lb` and `ub`.

Caution Do not use `@gacreationuniform` when you have linear constraints. Otherwise, your population might not satisfy the linear constraints.

- `Feasible population` (`@gacreationlinearfeasible`), the default when there are linear constraints and no integer constraints, creates a random initial population that satisfies all bounds and linear constraints. If there are linear constraints, `Feasible population` creates many individuals on the boundaries of the constraint region, and creates a well-dispersed population. `Feasible population` ignores **Initial range** (`PopInitRange`).

`gacreationlinearfeasible` calls `linprog` to create a feasible population with respect to bounds and linear constraints.

For an example showing its behavior, see "Linearly Constrained Population and Custom Plot Function" on page 5-77.

- `Nonlinear Feasible population` (`@gacreationnonlinearfeasible`) is the default creation function for the `'penalty'` nonlinear constraint algorithm. For details, see "Constraint Parameters" on page 10-47.
- `Custom` lets you write your own creation function, which must generate data of the type that you specify in **Population type**. To specify the creation function if you are using the Optimization app,

  - Set **Creation function** to `Custom`.
  - Set **Function name** to `@myfun`, where `myfun` is the name of your function.

  If you are using `ga`, set

  ```
  options = gaoptimset('CreationFcn', @myfun);
  ```

  Your creation function must have the following calling syntax.

  ```
  function Population = myfun(GenomeLength, FitnessFcn, options)
  ```

  The input arguments to the function are:

  - `Genomelength` — Number of independent variables for the fitness function
  - `FitnessFcn` — Fitness function
  - `options` — Options structure

  The function returns `Population`, the initial population for the genetic algorithm.

  "Passing Extra Parameters" in the Optimization Toolbox documentation explains how to provide additional parameters to the function.

Caution When you have bounds or linear constraints, ensure that your creation function creates individuals that satisfy these constraints. Otherwise, your population might not satisfy the constraints.

Initial population (`InitialPopulation`) specifies an initial population for the genetic algorithm. The default value is `[]`, in which case `ga` uses the default **Creation function** to create an initial population. If you enter a nonempty array in the **Initial population** field, the array must have no more than **Population size** rows, and exactly **Number of variables** columns. In this case, the genetic algorithm calls a **Creation function** to generate the remaining individuals, if required.

Initial scores (`InitialScores`) specifies initial scores for the initial population. The initial scores can also be partial. Do not specify initial scores with integer problems because `ga` overrides any choice you make.

Initial range (`PopInitRange`) specifies the range of the vectors in the initial population that is generated by the `gacreationuniform` creation function. You can set **Initial range** to be a matrix with two rows and **Number of variables** columns, each column of which has the form `[lb;ub]`, where `lb` is the lower bound and `ub` is the upper bound for the entries in that coordinate. If you specify **Initial range** to be a 2-by-1 vector, each entry is expanded to a constant row of length **Number of variables**. If you do not specify an **Initial range**, the default is `[-10;10]` (`[-1e4+1;1e4+1]` for integer-constrained problems), modified to match any existing bounds.

See “Setting the Initial Range” on page 5-73 for an example.

Fitness Scaling Options

Fitness scaling converts the raw fitness scores that are returned by the fitness function to values in a range that is suitable for the selection function. You can specify options for fitness scaling in the **Fitness scaling** pane.

Scaling function (`FitnessScalingFcn`) specifies the function that performs the scaling. The options are

- `Rank` (`@fitscalingrank`) — The default fitness scaling function, `Rank`, scales the raw scores based on the rank of each individual instead of its score. The rank of an individual is its position in the sorted scores. An individual with rank r has scaled score proportional to $1/\sqrt{r}$. So the scaled score of the most fit individual is proportional to 1, the scaled score of the next most fit is proportional to $1/\sqrt{2}$, and so on. Rank fitness scaling removes the effect of the spread of the raw scores. The square root makes poorly ranked individuals more nearly equal in score, compared to rank scoring. For more information, see “Fitness Scaling” on page 5-84.

- `Proportional` (`@fitscalingprop`) — Proportional scaling makes the scaled value of an individual proportional to its raw fitness score.
- `Top` (`@fitscalingtop`) — Top scaling scales the top individuals equally. Selecting `Top` displays an additional field, **Quantity**, which specifies the number of individuals that are assigned positive scaled values. **Quantity** can be an integer from 1 through the population size or a fraction from 0 through 1 specifying a fraction of the population size. The default value is `0.4`. Each of the individuals that produce offspring is assigned an equal scaled value, while the rest are assigned the value 0. The scaled values have the form [01/n 1/n 0 0 1/n 0 0 1/n ...].

  To change the default value for **Quantity** at the command line, use the following syntax:

  ```
  options = gaoptimset('FitnessScalingFcn', {@fitscalingtop,quantity})
  ```

  where `quantity` is the value of **Quantity**.
- `Shift linear` (`@fitscalingshiftlinear`) — Shift linear scaling scales the raw scores so that the expectation of the fittest individual is equal to a constant multiplied by the average score. You specify the constant in the **Max survival rate** field, which is displayed when you select `Shift linear`. The default value is `2`.

  To change the default value of **Max survival rate** at the command line, use the following syntax

  ```
  options = gaoptimset('FitnessScalingFcn',
  {@fitscalingshiftlinear, rate})
  ```

  where `rate` is the value of **Max survival rate**.
- `Custom` lets you write your own scaling function. To specify the scaling function using the Optimization app,

  - Set **Scaling function** to `Custom`.
  - Set **Function name** to `@myfun`, where `myfun` is the name of your function.

  If you are using `ga` at the command line, set

```
options = gaoptimset('FitnessScalingFcn', @myfun);
```

Your scaling function must have the following calling syntax:

```
function expectation = myfun(scores, nParents)
```

The input arguments to the function are:

- `scores` — A vector of scalars, one for each member of the population
- `nParents` — The number of parents needed from this population

The function returns `expectation`, a column vector of scalars of the same length as `scores`, giving the scaled values of each member of the population. The sum of the entries of `expectation` must equal `nParents`.

"Passing Extra Parameters" in the Optimization Toolbox documentation explains how to provide additional parameters to the function.

See "Fitness Scaling" on page 5-84 for more information.

Selection Options

Selection options specify how the genetic algorithm chooses parents for the next generation. You can specify the function the algorithm uses in the **Selection function** (`SelectionFcn`) field in the **Selection** options pane. Do not use with integer problems. The options are

- `Stochastic uniform` (`@selectionstochunif`) — The default selection function, `Stochastic uniform`, lays out a line in which each parent corresponds to a section of the line of length proportional to its scaled value. The algorithm moves along the line in steps of equal size. At each step, the algorithm allocates a parent from the section it lands on. The first step is a uniform random number less than the step size.
- `Remainder` (`@selectionremainder`) — Remainder selection assigns parents deterministically from the integer part of each individual's scaled value and then uses roulette selection on the remaining fractional part. For example, if the scaled value of an individual is 2.3, that individual is listed twice as a parent because the integer part is 2. After parents have been assigned according to the integer parts of the scaled values, the rest of the parents are chosen stochastically. The probability that a parent is chosen in this step is proportional to the fractional part of its scaled value.

- `Uniform` (`@selectionuniform`) — Uniform selection chooses parents using the expectations and number of parents. Uniform selection is useful for debugging and testing, but is not a very effective search strategy.
- `Roulette` (`@selectionroulette`) — Roulette selection chooses parents by simulating a roulette wheel, in which the area of the section of the wheel corresponding to an individual is proportional to the individual's expectation. The algorithm uses a random number to select one of the sections with a probability equal to its area.
- `Tournament` (`@selectiontournament`) — Tournament selection chooses each parent by choosing **Tournament size** players at random and then choosing the best individual out of that set to be a parent. **Tournament size** must be at least 2. The default value of **Tournament size** is `4`.

  To change the default value of **Tournament size** at the command line, use the syntax

  ```
  options = gaoptimset('SelectionFcn',...
                       {@selectiontournament,size})
  ```

  where `size` is the value of **Tournament size**.

  When **Constraint parameters > Nonlinear constraint algorithm** is `Penalty`, `ga` uses `Tournament` with size `2`.
- `Custom` enables you to write your own selection function. To specify the selection function using the Optimization app,

  - Set **Selection function** to `Custom`.
  - Set **Function name** to `@myfun`, where `myfun` is the name of your function.

  If you are using `ga` at the command line, set

  ```
  options = gaoptimset('SelectionFcn', @myfun);
  ```

  Your selection function must have the following calling syntax:

  ```
  function parents = myfun(expectation, nParents, options)
  ```

  The input arguments to the function are

  - `expectation` — Expected number of children for each member of the population
  - `nParents`— Number of parents to select

- `options` — Genetic algorithm `options` structure

  The function returns `parents`, a row vector of length `nParents` containing the indices of the parents that you select.

  "Passing Extra Parameters" in the Optimization Toolbox documentation explains how to provide additional parameters to the function.

See "Selection" on page 5-25 for more information.

Reproduction Options

Reproduction options specify how the genetic algorithm creates children for the next generation.

Elite count (`EliteCount`) specifies the number of individuals that are guaranteed to survive to the next generation. Set **Elite count** to be a positive integer less than or equal to the population size. The default value is `ceil(0.05*PopulationSize)` for continuous problems, and `0.05*(default PopulationSize)` for mixed-integer problems.

Crossover fraction (`CrossoverFraction`) specifies the fraction of the next generation, other than elite children, that are produced by crossover. Set **Crossover fraction** to be a fraction between `0` and `1`, either by entering the fraction in the text box or moving the slider. The default value is `0.8`.

See "Setting the Crossover Fraction" on page 5-90 for an example.

Mutation Options

Mutation options specify how the genetic algorithm makes small random changes in the individuals in the population to create mutation children. Mutation provides genetic diversity and enables the genetic algorithm to search a broader space. You can specify the mutation function in the **Mutation function** (`MutationFcn`) field in the **Mutation** options pane. Do not use with integer problems. You can choose from the following functions:

- `Gaussian` (`mutationgaussian`) — The default mutation function for unconstrained problems, `Gaussian`, adds a random number taken from a Gaussian distribution with mean 0 to each entry of the parent vector. The standard deviation of this distribution is determined by the parameters **Scale** and **Shrink**, which are displayed when you select `Gaussian`, and by the **Initial range** setting in the **Population** options.

- The **Scale** parameter determines the standard deviation at the first generation. If you set **Initial range** to be a 2-by-1 vector `v`, the initial standard deviation is the same at all coordinates of the parent vector, and is given by **Scale**`*(v(2)-v(1))`.

  If you set **Initial range** to be a vector `v` with two rows and **Number of variables** columns, the initial standard deviation at coordinate `i` of the parent vector is given by **Scale**`*(v(i,2) - v(i,1))`.
- The **Shrink** parameter controls how the standard deviation shrinks as generations go by. If you set **Initial range** to be a 2-by-1 vector, the standard deviation at the *k*th generation, σ_k, is the same at all coordinates of the parent vector, and is given by the recursive formula

$$\sigma_k = \sigma_{k-1}\left(1 - \text{Shrink}\frac{k}{\text{Generations}}\right).$$

  If you set **Initial range** to be a vector with two rows and **Number of variables** columns, the standard deviation at coordinate *i* of the parent vector at the *k*th generation, $\sigma_{i,k}$, is given by the recursive formula

$$\sigma_{i,k} = \sigma_{i,k-1}\left(1 - \text{Shrink}\frac{k}{\text{Generations}}\right).$$

  If you set **Shrink** to `1`, the algorithm shrinks the standard deviation in each coordinate linearly until it reaches 0 at the last generation is reached. A negative value of **Shrink** causes the standard deviation to grow.

The default value of both **Scale** and **Shrink** is 1. To change the default values at the command line, use the syntax

```
options = gaoptimset('MutationFcn', ...
{@mutationgaussian, scale, shrink})
```

where `scale` and `shrink` are the values of **Scale** and **Shrink**, respectively.

Caution Do not use `mutationgaussian` when you have bounds or linear constraints. Otherwise, your population will not necessarily satisfy the constraints.

- `Uniform` (`mutationuniform`) — Uniform mutation is a two-step process. First, the algorithm selects a fraction of the vector entries of an individual for mutation, where each entry has a probability **Rate** of being mutated. The default value of **Rate** is `0.01`. In the second step, the algorithm replaces each selected entry by a random number selected uniformly from the range for that entry.

  To change the default value of **Rate** at the command line, use the syntax

  ```
  options = gaoptimset('MutationFcn', {@mutationuniform, rate})
  ```

  where `rate` is the value of **Rate**.

  Caution Do not use `mutationuniform` when you have bounds or linear constraints. Otherwise, your population will not necessarily satisfy the constraints.

- `Adaptive Feasible` (`mutationadaptfeasible`), the default mutation function when there are constraints, randomly generates directions that are adaptive with respect to the last successful or unsuccessful generation. The mutation chooses a direction and step length that satisfies bounds and linear constraints.
- `Custom` enables you to write your own mutation function. To specify the mutation function using the Optimization app,

  - Set **Mutation function** to `Custom`.
  - Set **Function name** to `@myfun`, where `myfun` is the name of your function.

  If you are using `ga`, set

  ```
  options = gaoptimset('MutationFcn', @myfun);
  ```

  Your mutation function must have this calling syntax:

  ```
  function mutationChildren = myfun(parents, options, nvars,
  FitnessFcn, state, thisScore, thisPopulation)
  ```

  The arguments to the function are

  - `parents` — Row vector of parents chosen by the selection function

- `options` — Options structure
- `nvars` — Number of variables
- `FitnessFcn` — Fitness function
- `state` — Structure containing information about the current generation. “The State Structure” on page 10-31 describes the fields of `state`.
- `thisScore` — Vector of scores of the current population
- `thisPopulation` — Matrix of individuals in the current population

The function returns `mutationChildren`—the mutated offspring—as a matrix where rows correspond to the children. The number of columns of the matrix is **Number of variables**.

“Passing Extra Parameters” in the Optimization Toolbox documentation explains how to provide additional parameters to the function.

Caution When you have bounds or linear constraints, ensure that your mutation function creates individuals that satisfy these constraints. Otherwise, your population will not necessarily satisfy the constraints.

Crossover Options

Crossover options specify how the genetic algorithm combines two individuals, or parents, to form a crossover child for the next generation.

Crossover function (`CrossoverFcn`) specifies the function that performs the crossover. Do not use with integer problems. You can choose from the following functions:

- `Scattered` (`@crossoverscattered`), the default crossover function for problems without linear constraints, creates a random binary vector and selects the genes where the vector is a 1 from the first parent, and the genes where the vector is a 0 from the second parent, and combines the genes to form the child. For example, if `p1` and `p2` are the parents

  ```
  p1 = [a b c d e f g h]
  p2 = [1 2 3 4 5 6 7 8]
  ```

  and the binary vector is [1 1 0 0 1 0 0 0], the function returns the following child:

  ```
  child1 = [a b 3 4 e 6 7 8]
  ```

Caution Do not use `@crossoverscattered` when you have linear constraints. Otherwise, your population will not necessarily satisfy the constraints.

- `Single point` (`@crossoversinglepoint`) chooses a random integer n between 1 and **Number of variables** and then

  - Selects vector entries numbered less than or equal to n from the first parent.
  - Selects vector entries numbered greater than n from the second parent.
  - Concatenates these entries to form a child vector.

    For example, if `p1` and `p2` are the parents

    ```
    p1 = [a b c d e f g h]
    p2 = [1 2 3 4 5 6 7 8]
    ```

  and the crossover point is 3, the function returns the following child.

  ```
  child = [a b c 4 5 6 7 8]
  ```

Caution Do not use `@crossoversinglepoint` when you have linear constraints. Otherwise, your population will not necessarily satisfy the constraints.

- `Two point` (`@crossovertwopoint`) selects two random integers `m` and `n` between `1` and **Number of variables**. The function selects

  - Vector entries numbered less than or equal to `m` from the first parent
  - Vector entries numbered from `m+1` to `n`, inclusive, from the second parent
  - Vector entries numbered greater than `n` from the first parent.

  The algorithm then concatenates these genes to form a single gene. For example, if `p1` and `p2` are the parents

  ```
  p1 = [a b c d e f g h]
  p2 = [1 2 3 4 5 6 7 8]
  ```

  and the crossover points are 3 and 6, the function returns the following child.

  ```
  child = [a b c 4 5 6 g h]
  ```

Caution Do not use `@crossovertwopoint` when you have linear constraints. Otherwise, your population will not necessarily satisfy the constraints.

- `Intermediate` (`@crossoverintermediate`), the default crossover function when there are linear constraints, creates children by taking a weighted average of the parents. You can specify the weights by a single parameter, **Ratio**, which can be a scalar or a row vector of length **Number of variables**. The default is a vector of all 1's. The function creates the child from `parent1` and `parent2` using the following formula.

  ```
  child = parent1 + rand * Ratio * ( parent2 - parent1)
  ```

  If all the entries of **Ratio** lie in the range [0, 1], the children produced are within the hypercube defined by placing the parents at opposite vertices. If **Ratio** is not in that range, the children might lie outside the hypercube. If **Ratio** is a scalar, then all the children lie on the line between the parents.

  To change the default value of **Ratio** at the command line, use the syntax

  ```
  options = gaoptimset('CrossoverFcn', ...
  {@crossoverintermediate, ratio});
  ```

  where `ratio` is the value of **Ratio**.

- `Heuristic` (`@crossoverheuristic`) returns a child that lies on the line containing the two parents, a small distance away from the parent with the better fitness value in the direction away from the parent with the worse fitness value. You can specify how far the child is from the better parent by the parameter **Ratio**, which appears when you select `Heuristic`. The default value of **Ratio** is 1.2. If `parent1` and `parent2` are the parents, and `parent1` has the better fitness value, the function returns the child

  ```
  child = parent2 + R * (parent1 - parent2);
  ```

  To change the default value of **Ratio** at the command line, use the syntax

  ```
  options=gaoptimset('CrossoverFcn',...
                     {@crossoverheuristic,ratio});
  ```

  where `ratio` is the value of **Ratio**.

- `Arithmetic` (`@crossoverarithmetic`) creates children that are the weighted arithmetic mean of two parents. Children are always feasible with respect to linear constraints and bounds.
- `Custom` enables you to write your own crossover function. To specify the crossover function using the Optimization app,

  - Set **Crossover function** to `Custom`.
  - Set **Function name** to `@myfun`, where `myfun` is the name of your function.

  If you are using `ga`, set

  ```
  options = gaoptimset('CrossoverFcn',@myfun);
  ```

  Your crossover function must have the following calling syntax.

  ```
  xoverKids = myfun(parents, options, nvars, FitnessFcn, ...
      unused,thisPopulation)
  ```

  The arguments to the function are

  - `parents` — Row vector of parents chosen by the selection function
  - `options` — `options` structure
  - `nvars` — Number of variables
  - `FitnessFcn` — Fitness function
  - `unused` — Placeholder not used
  - `thisPopulation` — Matrix representing the current population. The number of rows of the matrix is **Population size** and the number of columns is **Number of variables**.

  The function returns `xoverKids`—the crossover offspring—as a matrix where rows correspond to the children. The number of columns of the matrix is **Number of variables**.

  "Passing Extra Parameters" in the Optimization Toolbox documentation explains how to provide additional parameters to the function.

  Caution When you have bounds or linear constraints, ensure that your crossover function creates individuals that satisfy these constraints. Otherwise, your population will not necessarily satisfy the constraints.

Migration Options

Note: *Subpopulations* refer to a form of parallel processing for the genetic algorithm. `ga` currently does not support this form. In subpopulations, each worker hosts a number of individuals. These individuals are a subpopulation. The worker evolves the subpopulation independently of other workers, except when migration causes some individuals to travel between workers.

Because `ga` does not currently support this form of parallel processing, there is no benefit to setting `PopulationSize` to a vector, or to setting the `MigrationDirection`, `MigrationInterval`, or `MigrationFraction` options.

Migration options specify how individuals move between subpopulations. Migration occurs if you set **Population size** to be a vector of length greater than 1. When migration occurs, the best individuals from one subpopulation replace the worst individuals in another subpopulation. Individuals that migrate from one subpopulation to another are copied. They are not removed from the source subpopulation.

You can control how migration occurs by the following three fields in the **Migration** options pane:

- **Direction** (`MigrationDirection`) — Migration can take place in one or both directions.

  - If you set **Direction** to `Forward` (`'forward'`), migration takes place toward the last subpopulation. That is, the *n*th subpopulation migrates into the (*n*+1)th subpopulation.
  - If you set **Direction** to `Both` (`'both'`), the n^{th} subpopulation migrates into both the (*n*–1)th and the (*n*+1)th subpopulation.

  Migration wraps at the ends of the subpopulations. That is, the last subpopulation migrates into the first, and the first may migrate into the last.
- **Interval** (`MigrationInterval`) — Specifies how many generation pass between migrations. For example, if you set **Interval** to `20`, migration takes place every 20 generations.
- **Fraction** (`MigrationFraction`) — Specifies how many individuals move between subpopulations. **Fraction** specifies the fraction of the smaller of the two subpopulations that moves. For example, if individuals migrate from a subpopulation

of 50 individuals into a subpopulation of 100 individuals and you set **Fraction** to `0.1`, the number of individuals that migrate is 0.1*50=5.

Constraint Parameters

Constraint parameters refer to the nonlinear constraint solver. For details on the algorithm, see "Nonlinear Constraint Solver Algorithms" on page 5-49.

Choose between the nonlinear constraint algorithms by setting the `NonlinConAlgorithm` option to `'auglag'` (Augmented Lagrangian) or `'penalty'` (Penalty algorithm).

- "Augmented Lagrangian Genetic Algorithm" on page 10-47
- "Penalty Algorithm" on page 10-47

Augmented Lagrangian Genetic Algorithm

- **Initial penalty** (`InitialPenalty`) — Specifies an initial value of the penalty parameter that is used by the nonlinear constraint algorithm. **Initial penalty** must be greater than or equal to `1`, and has a default of `10`.
- **Penalty factor** (`PenaltyFactor`) — Increases the penalty parameter when the problem is not solved to required accuracy and constraints are not satisfied. **Penalty factor** must be greater than `1`, and has a default of `100`.

Penalty Algorithm

The penalty algorithm uses the `gacreationnonlinearfeasible` creation function by default. This creation function uses `fmincon` to find feasible individuals. `gacreationnonlinearfeasible` starts `fmincon` from a variety of initial points within the bounds from the `PopInitRange` option. Optionally, `gacreationnonlinearfeasible` can run `fmincon` in parallel on the initial points.

You can specify tuning parameters for `gacreationnonlinearfeasible` using the following name-value pairs.

Name	Value
`SolverOpts`	`fmincon` options, created using `optimoptions` or `optimset`.
`UseParallel`	When `true`, run `fmincon` in parallel on initial points; default is `false`.

Name	Value
`NumStartPts`	Number of start points, a positive integer up to `sum(PopulationSize)` in value.

Include the name-value pairs in a cell array along with `@gacreationnonlinearfeasible`.

```
options = gaoptimset('CreationFcn',{@gacreationnonlinearfeasible,...
    'UseParallel',true,'NumStartPts',20});
```

Multiobjective Options

Multiobjective options define parameters characteristic of the multiobjective genetic algorithm. You can specify the following parameters:

- `DistanceMeasureFcn` — Defines a handle to the function that computes distance measure of individuals, computed in decision variable or design space (genotype) or in function space (phenotype). For example, the default distance measure function is `distancecrowding` in function space, or `{@distancecrowding,'phenotype'}`.
- `ParetoFraction` — Sets the fraction of individuals to keep on the first Pareto front while the solver selects individuals from higher fronts. This option is a scalar between 0 and 1.

Hybrid Function Options

A hybrid function is another minimization function that runs after the genetic algorithm terminates. You can specify a hybrid function in **Hybrid function** (`HybridFcn`) options. Do not use with integer problems. The choices are

- `[]` — No hybrid function.
- `fminsearch` (`@fminsearch`) — Uses the MATLAB function `fminsearch` to perform unconstrained minimization.
- `patternsearch` (`@patternsearch`) — Uses a pattern search to perform constrained or unconstrained minimization.
- `fminunc` (`@fminunc`) — Uses the Optimization Toolbox function `fminunc` to perform unconstrained minimization.
- `fmincon` (`@fmincon`) — Uses the Optimization Toolbox function `fmincon` to perform constrained minimization.

You can set separate options for the hybrid function. Use `optimset` for `fminsearch`, `psoptimset` for `patternsearch`, or `optimoptions` for `fmincon` or `fminunc`. For example:

```
hybridopts = optimoptions('fminunc','Display','iter','Algorithm','quasi-newton');
```

Include the hybrid options in the Genetic Algorithm `options` structure as follows:

```
options = gaoptimset(options,'HybridFcn',{@fminunc,hybridopts});
```

`hybridopts` must exist before you set `options`.

See “Include a Hybrid Function” on page 5-104 for an example.

Stopping Criteria Options

Stopping criteria determine what causes the algorithm to terminate. You can specify the following options:

- **Generations** (`Generations`) — Specifies the maximum number of iterations for the genetic algorithm to perform. The default is `100*numberOfVariables`.
- **Time limit** (`TimeLimit`) — Specifies the maximum time in seconds the genetic algorithm runs before stopping, as measured by `cputime`.
- **Fitness limit** (`FitnessLimit`) — The algorithm stops if the best fitness value is less than or equal to the value of **Fitness limit**.
- **Stall generations** (`StallGenLimit`) — The algorithm stops if the average relative change in the best fitness function value over **Stall generations** is less than or equal to **Function tolerance**. (If the `StallTest` option is `'geometricWeighted'`, then the test is for a *geometric weighted* average relative change.) For a problem with nonlinear constraints, **Stall generations** applies to the subproblem (see “Nonlinear Constraint Solver Algorithms” on page 5-49).

  For `gamultiobj`, if the weighted average relative change in the *spread* of the Pareto solutions over **Stall generations** is less than **Function tolerance**, and the spread is smaller than the average spread over the last **Stall generations**, then the algorithm stops. The *spread* is a measure of the movement of the Pareto front.
- **Stall time limit** (`StallTimeLimit`) — The algorithm stops if there is no improvement in the best fitness value for an interval of time in seconds specified by **Stall time limit**, as measured by `cputime`.
- **Function tolerance** (`TolFun`) — The algorithm stops if the average relative change in the best fitness function value over **Stall generations** is less than or equal to

Function tolerance. (If the `StallTest` option is `'geometricWeighted'`, then the test is for a *geometric weighted* average relative change.)

For `gamultiobj`, if the weighted average relative change in the *spread* of the Pareto solutions over **Stall generations** is less than **Function tolerance**, and the spread is smaller than the average spread over the last **Stall generations**, then the algorithm stops. The *spread* is a measure of the movement of the Pareto front.

- **Constraint tolerance** (`TolCon`) — The **Constraint tolerance** is not used as stopping criterion. It is used to determine the feasibility with respect to nonlinear constraints. Also, `max(sqrt(eps),sqrt(TolCon))` determines feasibility with respect to linear constraints.

See “Set Maximum Number of Generations” on page 5-108 for an example.

Output Function Options

Output functions are functions that the genetic algorithm calls at each generation. To specify the output function using the Optimization app,

- Select **Custom function**.
- Enter `@myfun` in the text box, where `myfun` is the name of your function. Write `myfun` with appropriate syntax.
- To pass extra parameters in the output function, use “Anonymous Functions”.
- For multiple output functions, enter a cell array of output function handles: `{@myfun1,@myfun2,...}`.

At the command line, set

```
options = gaoptimset('OutputFcns',@myfun);
```

For multiple output functions, enter a cell array:

```
options = gaoptimset('OutputFcns',{@myfun1,@myfun2,...});
```

To see a template that you can use to write your own output functions, enter

```
edit gaoutputfcntemplate
```

at the MATLAB command line.

Structure of the Output Function

The output function has the following calling syntax.

```
[state,options,optchanged] = myfun(options,state,flag)
```

The function has the following input arguments:

- `options` — Options structure
- `state` — Structure containing information about the current generation. “The State Structure” on page 10-31 describes the fields of `state`.
- `flag` — String indicating the current status of the algorithm as follows:
  - `'init'` — Initial stage
  - `'iter'` — Algorithm running
  - `'interrupt'` — Intermediate stage
  - `'done'` — Algorithm terminated

“Passing Extra Parameters” in the Optimization Toolbox documentation explains how to provide additional parameters to the function.

The output function returns the following arguments to `ga`:

- `state` — Structure containing information about the current generation. “The State Structure” on page 10-31 describes the fields of `state`. To stop the iterations, set `state.StopFlag` to a nonempty string.
- `options` — Options structure modified by the output function. This argument is optional.
- `optchanged` — Flag indicating changes to `options`

Display to Command Window Options

Level of display (`'Display'`) specifies how much information is displayed at the command line while the genetic algorithm is running. The available options are

- `Off` (`'off'`) — No output is displayed.
- `Iterative` (`'iter'`) — Information is displayed at each iteration.
- `Diagnose` (`'diagnose'`) — Information is displayed at each iteration. In addition, the diagnostic lists some problem information and the options that have been changed from the defaults.
- `Final` (`'final'`) — The reason for stopping is displayed.

Both `Iterative` and `Diagnose` display the following information:

- `Generation` — Generation number
- `f-count` — Cumulative number of fitness function evaluations
- `Best f(x)` — Best fitness function value
- `Mean f(x)` — Mean fitness function value
- `Stall generations` — Number of generations since the last improvement of the fitness function

When a nonlinear constraint function has been specified, `Iterative` and `Diagnose` do not display the `Mean f(x)`, but will additionally display:

- `Max Constraint` — Maximum nonlinear constraint violation

The default value of **Level of display** is

- `Off` in the Optimization app
- `'final'` in an options structure created using `gaoptimset`

Vectorize and Parallel Options (User Function Evaluation)

You can choose to have your fitness and constraint functions evaluated in serial, parallel, or in a vectorized fashion. These options are available in the **User function evaluation** section of the **Options** pane of the Optimization app, or by setting the `'Vectorized'` and `'UseParallel'` options with `gaoptimset`.

- When **Evaluate fitness and constraint functions** (`'Vectorized'`) is **in serial** (`'off'`), `ga` calls the fitness function on one individual at a time as it loops through the population. (At the command line, this assumes `'UseParallel'` is at its default value of `false`.)
- When **Evaluate fitness and constraint functions** (`'Vectorized'`) is **vectorized** (`'on'`), `ga` calls the fitness function on the entire population at once, i.e., in a single call to the fitness function.

  If there are nonlinear constraints, the fitness function and the nonlinear constraints all need to be vectorized in order for the algorithm to compute in a vectorized manner.

  See “Vectorize the Fitness Function” on page 5-110 for an example.
- When **Evaluate fitness and constraint functions** (`UseParallel`) is **in parallel** (`true`), `ga` calls the fitness function in parallel, using the parallel environment you

established (see “How to Use Parallel Processing” on page 9-12). At the command line, set `UseParallel` to `false` to compute serially.

Note: You cannot simultaneously use vectorized and parallel computations. If you set `'UseParallel'` to `true` and `'Vectorized'` to `'on'`, `ga` evaluates your fitness and constraint functions in a vectorized manner, not in parallel.

How Fitness and Constraint Functions Are Evaluated

	`Vectorized = 'Off'`	**`Vectorized = 'On'`**
`UseParallel = false`	Serial	Vectorized
`UseParallel = true`	Parallel	Vectorized

Particle Swarm Options

In this section...

Specifying Options for particleswarm

Create `options` using the `optimoptions` function as follows.

```
options = optimoptions('particleswarm','Param1',value1,'Param2',value2,...);
```

For an example, see "Optimize Using Particle Swarm" on page 6-3.

Each option in this section is listed by its field name in `options`. For example, `Display` refers to the corresponding field of `options`.

Swarm Creation

By default, `particleswarm` calls the `@pswcreationuniform` swarm creation function. This function works as follows.

1. If an `InitialSwarm` option exists, `@pswcreationuniform` takes the first `SwarmSize` rows of the `InitialSwarm` matrix as the swarm. If the number of rows of the `InitialSwarm` matrix is smaller than `SwarmSize`, then `@pswcreationuniform` continues to the next step.
2. `@pswcreationuniform` creates enough particles so that there are `SwarmSize` in total. `@pswcreationuniform` creates particles that are randomly, uniformly distributed. The range for any swarm component is `-InitialSwarmSpan/2,InitialSwarmSpan/2`, shifted and scaled if necessary to match any bounds.

After creation, `particleswarm` checks that all particles satisfy any bounds, and truncates components if necessary. If the `Display` option is `'iter'` and a particle needed truncation, then `particleswarm` notifies you.

Custom Creation Function

Set a custom creation function using `optimoptions` to set the `CreationFcn` option to `@`*`customcreation`*, where *`customcreation`* is the name of your creation function file. A custom creation function has this syntax.

```
swarm = customcreation(problem)
```

The creation function should return a matrix of size `SwarmSize`-by-`nvars`, where each row represents the location of one particle. See `problem` for details of the problem structure. In particular, you can obtain `SwarmSize` from `problem.options.SwarmSize`, and `nvars` from `problem.nvars`.

For an example of a creation function, see the code for `pswcreationuniform`.

```
edit pswcreationuniform
```

Display Settings

The `Display` option specifies how much information is displayed at the command line while the algorithm is running.

- `'off'` or `'none'` — No output is displayed.
- `'iter'` — Information is displayed at each iteration.
- `'final'` (default) — The reason for stopping is displayed.

`iter` displays:

- `Iteration` — Iteration number
- `f-count` — Cumulative number of objective function evaluations
- `Best f(x)` — Best objective function value
- `Mean f(x)` — Mean objective function value over all particles
- `Stall Iterations` — Number of iterations since the last change in `Best f(x)`

The `DisplayInterval` option sets the number of iterations that are performed before the iterative display updates. Give a positive integer.

Algorithm Settings

The details of the `particleswarm` algorithm appear in “Particle Swarm Optimization Algorithm” on page 6-10. This section describes the tuning parameters.

The main step in the particle swarm algorithm is the generation of new velocities for the swarm:

For `u1` and `u2` uniformly (0,1) distributed random vectors of length `nvars`, update the velocity
`v = W*v + y1*u1.*(p-x) + y2*u2.*(g-x)`.

The variables `W = inertia`, `y1 = SelfAdjustment`, and `y2 = SocialAdjustment`.

This update uses a weighted sum of:

- The previous velocity `v`
- `x-p`, the difference between the current position `x` and the best position `p` the particle has seen
- `x-g`, the difference between the current position `x` and the best position `g` in the current neighborhood

Based on this formula, the options have the following effect:

- Larger absolute value of inertia `W` leads to the new velocity being more in the same line as the old, and with a larger absolute magnitude. A large absolute value of `W` can destabilize the swarm. The value of `W` stays within the range of the two-element vector `InertiaRange`.
- Larger values of `y1 = SelfAdjustment` make the particle head more toward the best place it has visited.
- Larger values of `y2 = SocialAdjustment` make the particle head more toward the best place in the current neighborhood.

Large values of inertia, `SelfAdjustment`, or `SocialAdjustment` can destabilize the swarm.

The `MinFractionNeighbors` option sets both the initial neighborhood size for each particle, and the minimum neighborhood size; see “Particle Swarm Optimization Algorithm” on page 6-10. Setting `MinFractionNeighbors` to `1` has all members of the swarm use the global minimum point as their societal adjustment target.

See “Optimize Using Particle Swarm” on page 6-3 for an example that sets a few of these tuning options.

Hybrid Function

A hybrid function is another minimization function that runs after the particle swarm algorithm terminates. You can specify a hybrid function in the `HybridFcn` option. The choices are

- `[]` — No hybrid function.
- `fminsearch` (`@fminsearch`) — Use the MATLAB function `fminsearch` to perform unconstrained minimization.
- `patternsearch` (`@patternsearch`) — Use a pattern search to perform constrained or unconstrained minimization.
- `fminunc` (`@fminunc`) — Use the Optimization Toolbox function `fminunc` to perform unconstrained minimization.
- `fmincon` (`@fmincon`) — Use the Optimization Toolbox function `fmincon` to perform constrained minimization.

You can set separate options for the hybrid function. Use `optimset` for `fminsearch`, `psoptimset` for `patternsearch`, or `optimoptions` for `fmincon` or `fminunc`. For example:

```
hybridopts = optimoptions('fminunc','Display','iter','Algorithm','quasi-newton');
```

Include the hybrid options in the `particleswarm` `options` as follows:

```
options = optimoptions(options,'HybridFcn',{@fminunc,hybridopts});
```

`hybridopts` must exist before you set `options`.

For an example that uses a hybrid function, see “Optimize Using Particle Swarm” on page 6-3.

Output Function and Plot Function

Output functions are functions that `particleswarm` calls at each iteration. Output functions can halt `particleswarm`, or can perform other tasks. To specify an output function,

```
options = optimoptions(@particleswarm,'OutputFcns',@outfun)
```

where `outfun` is a function with syntax specified in “Structure of the Output Function or Plot Function” on page 10-58. If you have several output functions, pass them in a cell array:

```
options = optimoptions(@particleswarm,'OutputFcns',{@outfun1,@outfun2,@outfun3})
```

Similarly, plot functions are functions that `particleswarm` calls at each iteration. The difference between an output function and a plot function is that a plot function has built-in plotting enhancements, such as buttons that appear on the plot window to pause or stop `particleswarm`. To specify a plot function,

```
options = optimoptions(@particleswarm,'PlotFcns',@plotfun)
```

where `plotfun` is a function with syntax specified in “Structure of the Output Function or Plot Function” on page 10-58. If you have several plot functions, pass them in a cell array:

```
options = optimoptions(@particleswarm,'PlotFcns',{@plotfun1,@plotfun2,@plotfun3})
```

The lone built-in plot function `@pswplotbestf` plots the best objective function value against iterations.

For an example of a custom output function, see “Particle Swarm Output Function” on page 6-7.

Structure of the Output Function or Plot Function

An output function has the following calling syntax:

```
stop = myfun(optimValues,state)
```

If your function sets `stop` to `true`, iterations end. Set `stop` to `false` to have `particleswarm` continue to calculate.

The function has the following input arguments:

- `optimValues` — Structure containing information about the swarm in the current iteration. Details are in “optimValues Structure” on page 10-59.
- `state` — String giving the state of the current iteration.
    - `'init'` — The solver has not begun to iterate. Your output function or plot function can use this state to open files, or set up data structures or plots for subsequent iterations.

- `'iter'` — The solver is proceeding with its iterations. Typically, this is where your output function or plot function performs its work.
- `'done'` — The solver reached a stopping criterion. Your output function or plot function can use this state to clean up, such as closing any files it opened.

"Passing Extra Parameters" in the Optimization Toolbox documentation explains how to provide additional parameters to output functions or plot functions.

optimValues Structure

`particleswarm` passes the `optimValues` structure to your output functions or plot functions. The `optimValues` structure has the following fields.

Field	Contents
`funccount`	Total number of objective function evaluations.
`bestx`	Best solution point found, corresponding to the best objective function value `bestfval`.
`bestfval`	Best (lowest) objective function value found.
`iteration`	Iteration number.
`meanfval`	Mean objective function among all particles at the current iteration.
`stalliterations`	Number of iterations since the last change in `bestfval`.
`swarm`	Matrix containing the particle positions. Each row contains the position of one particle, and the number of rows is equal to the swarm size.
`swarmfvals`	Vector containing the objective function values of particles in the swarm. For particle `i`, `swarmfvals(i) = fun(swarm(i,:))`, where `fun` is the objective function.

Parallel or Vectorized Function Evaluation

For increased speed, you can set your options so that `particleswarm` evaluates the objective function for the swarm in *parallel* or in a *vectorized* fashion. You can use only one of these options. If you set `UseParallel` to `true` and `Vectorized` to `'on'`, then the computations are done in a vectorized fashion, and not in parallel.

- "Parallel particleswarm" on page 10-60

- "Vectorized particleswarm" on page 10-60

Parallel particleswarm

If you have a Parallel Computing Toolbox license, you can distribute the evaluation of the objective functions to the swarm among your processors or cores. Set the `UseParallel` option to `true`.

Parallel computation is likely to be faster than serial when your objective function is computationally expensive, or when you have many particles and processors. Otherwise, communication overhead can cause parallel computation to be slower than serial computation.

For details, see "Parallel Computing".

Vectorized particleswarm

If your objective function can evaluate all the particles at once, you can usually save time by setting the `Vectorized` option to `'on'`. Your objective function should accept an `M`-by-`N` matrix, where each row represents one particle, and return an `M`-by-`1` vector of objective function values. This option works the same way as the `patternsearch` and `ga` `Vectorized` options. For `patternsearch` details, see "Vectorize the Objective and Constraint Functions" on page 4-80.

Stopping Criteria

`particleswarm` stops iterating when any of the following occur.

Stopping Option	Stopping Test	Exit Flag
`StallIterLimit` and `TolFun`	Relative change in the best objective function value `g` over the last `StallIterLimit` iterations is less than `TolFun`.	`1`
`MaxIter`	Number of iterations reaches `MaxIter`.	`0`
`OutputFcns` or `PlotFcns`	`OutputFcns` or `PlotFcns` can halt the iterations.	`-1`

Stopping Option	Stopping Test	Exit Flag
`ObjectiveLimit`	Best objective function value `g` is less than or equal to `ObjectiveLimit`.	`-3`
`StallTimeLimit`	Best objective function value `g` did not change in the last `StallTimeLimit` seconds.	`-4`
`MaxTime`	Function run time exceeds `MaxTime` seconds.	`-5`

Also, if you set the `FunValCheck` option to `'on'`, and the swarm has particles with `NaN`, `Inf`, or complex objective function values, `particleswarm` stops and issues an error.

Simulated Annealing Options

In this section...

saoptimset At The Command Line

Specify options by creating an `options` structure using the `saoptimset` function as follows:

```
options = saoptimset('Param1',value1,'Param2',value2, ...);
```

See "Set Options for simulannealbnd at the Command Line" on page 7-16 for examples.

Each option in this section is listed by its field name in the `options` structure. For example, `InitialTemperature` refers to the corresponding field of the `options` structure.

Plot Options

Plot options enable you to plot data from the simulated annealing solver while it is running.

`PlotInterval` specifies the number of iterations between consecutive calls to the plot function.

To display a plot when calling `simulannealbnd` from the command line, set the `PlotFcns` field of `options` to be a function handle to the plot function. You can specify any of the following plots:

- `@saplotbestf` plots the best objective function value.
- `@saplotbestx` plots the current best point.
- `@saplotf` plots the current function value.
- `@saplotx` plots the current point.
- `@saplotstopping` plots stopping criteria levels.
- `@saplottemperature` plots the temperature at each iteration.
- `@myfun` plots a custom plot function, where `myfun` is the name of your function. See "Structure of the Plot Functions" on page 10-11 for a description of the syntax.

For example, to display the best objective plot, set `options` as follows

```
options = saoptimset('PlotFcns',@saplotbestf);
```

To display multiple plots, use the cell array syntax

```
options = saoptimset('PlotFcns',{@plotfun1,@plotfun2, ...});
```

where `@plotfun1`, `@plotfun2`, and so on are function handles to the plot functions.

If you specify more than one plot function, all plots appear as subplots in the same window. Right-click any subplot to obtain a larger version in a separate figure window.

Structure of the Plot Functions

The first line of a plot function has the form

```
function stop = plotfun(options,optimvalues,flag)
```

The input arguments to the function are

- `options` — Options structure created using `saoptimset`.
- `optimvalues` — Structure containing information about the current state of the solver. The structure contains the following fields:
  - `x` — Current point
  - `fval` — Objective function value at x
  - `bestx` — Best point found so far
  - `bestfval` — Objective function value at best point
  - `temperature` — Current temperature

  - `iteration` — Current iteration
  - `funccount` — Number of function evaluations
  - `t0` — Start time for algorithm
  - `k` — Annealing parameter
- `flag` — Current state in which the plot function is called. The possible values for `flag` are
  - `'init'` — Initialization state
  - `'iter'` — Iteration state
  - `'done'` — Final state

The output argument `stop` provides a way to stop the algorithm at the current iteration. `stop` can have the following values:

- `false` — The algorithm continues to the next iteration.
- `true` — The algorithm terminates at the current iteration.

Temperature Options

Temperature options specify how the temperature will be lowered at each iteration over the course of the algorithm.

- `InitialTemperature` — Initial temperature at the start of the algorithm. The default is `100`. The initial temperature can be a vector with the same length as `x`, the vector of unknowns. `simulannealbnd` expands a scalar initial temperature into a vector.
- `TemperatureFcn` — Function used to update the temperature schedule. Let k denote the annealing parameter. (The annealing parameter is the same as the iteration number until reannealing.) The options are:
  - `@temperatureexp` — The temperature is equal to InitialTemperature * 0.95^k. This is the default.
  - `@temperaturefast` — The temperature is equal to InitialTemperature / k.
  - `@temperatureboltz` — The temperature is equal to InitialTemperature / ln(k).
  - `@myfun` — Uses a custom function, `myfun`, to update temperature. The syntax is:

    ```
    temperature = myfun(optimValues,options)
    ```

where `optimValues` is a structure described in "Structure of the Plot Functions" on page 10-63. `options` is either the structure created with `saoptimset`, or the structure of default options, if you did not create an options structure. Both the annealing parameter `optimValues.k` and the temperature `optimValues.temperature` are vectors with length equal to the number of elements of the current point `x`. For example, the function `temperaturefast` is:

```
temperature = options.InitialTemperature./optimValues.k;
```

Algorithm Settings

Algorithm settings define algorithmic specific parameters used in generating new points at each iteration.

Parameters that can be specified for `simulannealbnd` are:

- `DataType` — Type of data to use in the objective function. Choices:
  - `'double'` (default) — A vector of type `double`.
  - `'custom'` — Any other data type. You must provide a `'custom'` annealing function. You cannot use a hybrid function.
- `AnnealingFcn` — Function used to generate new points for the next iteration. The choices are:
  - `@annealingfast` — The step has length temperature, with direction uniformly at random. This is the default.
  - `@annealingboltz` — The step has length square root of temperature, with direction uniformly at random.
  - `@myfun` — Uses a custom annealing algorithm, `myfun`. The syntax is:

    ```
    newx = myfun(optimValues,problem)
    ```
    where `optimValues` is a structure described in "Structure of the Output Function" on page 10-68, and `problem` is a structure containing the following information:

    - `objective`: function handle to the objective function
    - `x0`: the start point
    - `nvar`: number of decision variables

    - `lb`: lower bound on decision variables
    - `ub`: upper bound on decision variables

  For example, the current position is `optimValues.x`, and the current objective function value is `problem.objective(optimValues.x)`.

- `ReannealInterval` — Number of points accepted before reannealing. The default value is `100`.
- `AcceptanceFcn` — Function used to determine whether a new point is accepted or not. The choices are:

    - `@acceptancesa` — Simulated annealing acceptance function, the default. If the new objective function value is less than the old, the new point is always accepted. Otherwise, the new point is accepted at random with a probability depending on the difference in objective function values and on the current temperature. The acceptance probability is

$$\frac{1}{1+\exp\left(\frac{\Delta}{\max(T)}\right)},$$

      where Δ = new objective – old objective, and T is the current temperature. Since both Δ and T are positive, the probability of acceptance is between 0 and 1/2. Smaller temperature leads to smaller acceptance probability. Also, larger Δ leads to smaller acceptance probability.
    - `@myfun` — A custom acceptance function, `myfun`. The syntax is:

```
acceptpoint = myfun(optimValues,newx,newfval);
```

      where `optimValues` is a structure described in "Structure of the Output Function" on page 10-68, `newx` is the point being evaluated for acceptance, and `newfval` is the objective function at `newx`. `acceptpoint` is a Boolean, with value `true` to accept `newx`, and `false` to reject `newx`.

Hybrid Function Options

A hybrid function is another minimization function that runs during or at the end of iterations of the solver. `HybridInterval` specifies the interval (if not `never` or `end`) at which the hybrid function is called. You can specify a hybrid function using the `HybridFcn` option. The choices are:

- `[]` — No hybrid function.
- `@fminsearch` — Uses the MATLAB function `fminsearch` to perform unconstrained minimization.
- `@patternsearch` — Uses `patternsearch` to perform constrained or unconstrained minimization.
- `@fminunc` — Uses the Optimization Toolbox function `fminunc` to perform unconstrained minimization.
- `@fmincon` — Uses the Optimization Toolbox function `fmincon` to perform constrained minimization.

You can set separate options for the hybrid function. Use `optimset` for `fminsearch`, `psoptimset` for `patternsearch`, or `optimoptions` for `fmincon` or `fminunc`. For example:

```
hybridopts = optimoptions('fminunc','Display','iter','Algorithm','quasi-newton');
```

Include the hybrid options in the `simulannealbnd` `options` structure as follows:

```
options = saoptimset(options,'HybridFcn',{@fminunc,hybridopts});
```

`hybridopts` must exist before you set `options`.

See "Include a Hybrid Function" on page 5-104 for an example.

Stopping Criteria Options

Stopping criteria determine what causes the algorithm to terminate. You can specify the following options:

- `TolFun` — The algorithm runs until the average change in value of the objective function in `StallIterLim` iterations is less than `TolFun`. The default value is `1e-6`.
- `MaxIter` — The algorithm stops if the number of iterations exceeds this maximum number of iterations. You can specify the maximum number of iterations as a positive integer or `Inf`. `Inf` is the default.
- `MaxFunEval` specifies the maximum number of evaluations of the objective function. The algorithm stops if the number of function evaluations exceeds the maximum number of function evaluations. The allowed maximum is `3000*numberofvariables`.
- `TimeLimit` specifies the maximum time in seconds the algorithm runs before stopping.

- `ObjectiveLimit` — The algorithm stops if the best objective function value is less than or equal to the value of `ObjectiveLimit`.

Output Function Options

Output functions are functions that the algorithm calls at each iteration. The default value is to have no output function, `[]`. You must first create an output function using the syntax described in “Structure of the Output Function” on page 10-68.

Using the Optimization app:

- Specify **Output function** as `@myfun`, where `myfun` is the name of your function.
- To pass extra parameters in the output function, use “Anonymous Functions”.
- For multiple output functions, enter a cell array of output function handles: `{@myfun1,@myfun2,...}`.

At the command line:

- `options = saoptimset('OutputFcns',@myfun);`
- For multiple output functions, enter a cell array:

  ```
  options = saoptimset('OutputFcns',{@myfun1,@myfun2,...});
  ```

To see a template that you can use to write your own output functions, enter

```
edit saoutputfcntemplate
```

at the MATLAB command line.

Structure of the Output Function

The output function has the following calling syntax.

```
[stop,options,optchanged] = myfun(options,optimvalues,flag)
```

The function has the following input arguments:

- `options` — Options structure created using `saoptimset`.
- `optimvalues` — Structure containing information about the current state of the solver. The structure contains the following fields:
  - `x` — Current point

  - `fval` — Objective function value at x
  - `bestx` — Best point found so far
  - `bestfval` — Objective function value at best point
  - `temperature` — Current temperature, a vector the same length as `x`
  - `iteration` — Current iteration
  - `funccount` — Number of function evaluations
  - `t0` — Start time for algorithm
  - `k` — Annealing parameter, a vector the same length as `x`
- `flag` — Current state in which the output function is called. The possible values for `flag` are
  - `'init'` — Initialization state
  - `'iter'` — Iteration state
  - `'done'` — Final state

"Passing Extra Parameters" in the Optimization Toolbox documentation explains how to provide additional parameters to the output function.

The output function returns the following arguments:

- `stop` — Provides a way to stop the algorithm at the current iteration. `stop` can have the following values:
  - `false` — The algorithm continues to the next iteration.
  - `true` — The algorithm terminates at the current iteration.
- `options` — Options structure modified by the output function.
- `optchanged` — A boolean flag indicating changes were made to `options`. This must be set to `true` if options are changed.

Display Options

Use the `Display` option to specify how much information is displayed at the command line while the algorithm is running. The available options are

- `off` — No output is displayed. This is the default value for an options structure created using `saoptimset`.

- `iter` — Information is displayed at each iteration.
- `diagnose` — Information is displayed at each iteration. In addition, the diagnostic lists some problem information and the options that have been changed from the defaults.
- `final` — The reason for stopping is displayed. This is the default.

Both `iter` and `diagnose` display the following information:

- `Iteration` — Iteration number
- `f-count` — Cumulative number of objective function evaluations
- `Best f(x)` — Best objective function value
- `Current f(x)` — Current objective function value
- `Mean Temperature` — Mean temperature function value

Functions — Alphabetical List

createOptimProblem

Create optimization problem structure

Syntax

```
problem = createOptimProblem('solverName')
problem =
createOptimProblem('solverName','ParameterName',ParameterValue,...)
```

Description

`problem = createOptimProblem('solverName')` creates an empty optimization problem structure for the `solverName` solver.

`problem =`
`createOptimProblem('solverName','ParameterName',ParameterValue,...)` accepts one or more comma-separated parameter name/value pairs. Specify `ParameterName` inside single quotes.

Input Arguments

solverName

Name of the solver. For a `GlobalSearch` problem, use `'fmincon'`. For a `MultiStart` problem, use `'fmincon'`, `'fminunc'`, `'lsqcurvefit'` or `'lsqnonlin'`.

Parameter Name/Value Pairs

'Aeq'

Matrix for linear equality constraints. The constraints have the form:

`Aeq x = beq`

`'Aineq'`

Matrix for linear inequality constraints. The constraints have the form:

$\texttt{Aineq}\ x \leq \texttt{bineq}$

`'beq'`

Vector for linear equality constraints. The constraints have the form:

$\texttt{Aeq}\ x = \texttt{beq}$

`'bineq'`

Vector for linear inequality constraints. The constraints have the form:

$\texttt{Aineq}\ x \leq \texttt{bineq}$

`'lb'`

Vector of lower bounds.

`lb` can also be an array; see “Matrix Arguments”.

`'nonlcon'`

Function handle to the nonlinear constraint function. The constraint function must accept a vector `x` and return two vectors: `c`, the nonlinear inequality constraints, and `ceq`, the nonlinear equality constraints. If one of these constraint functions is empty, `nonlcon` must return `[]` for that function.

If the `GradConstr` option is `'on'`, then in addition `nonlcon` must return two additional outputs, `gradc` and `gradceq`. The `gradc` parameter is a matrix with one column for the gradient of each constraint, as is `gradceq`.

For more information, see “Write Constraints” on page 2-6.

`'objective'`

Function handle to the objective function. For all solvers except `lsqnonlin` and `lsqcurvefit`, the objective function must accept a vector `x` and return a scalar. If the `GradObj` option is `'on'`, then the objective function must return a second output, a vector, representing the gradient of the objective. For `lsqnonlin`, the objective function must accept a vector `x` and return a vector. `lsqnonlin` sums the squares of the objective function values. For `lsqcurvefit`, the objective function must accept two inputs, `x` and `xdata`, and return a vector.

For more information, see “Compute Objective Functions” on page 2-2.

`'options'`

Optimization options. Create options with `optimoptions`, or by exporting from the Optimization app.

`'ub'`

Vector of upper bounds.

`ub` can also be an array; see “Matrix Arguments”.

`'x0'`

A vector, a potential starting point for the optimization. Gives the dimensionality of the problem.

`x0` can also be an array; see “Matrix Arguments”.

`'xdata'`

Vector of data points for `lsqcurvefit`.

`'ydata'`

Vector of data points for `lsqcurvefit`.

Output Arguments

`problem`

Optimization problem structure.

Examples

Create a problem structure using Rosenbrock's function as objective (see “Include a Hybrid Function” on page 5-104), the `interior-point` algorithm for `fmincon`, and bounds with absolute value `2`:

```
anonrosen = @(x)(100*(x(2) - x(1)^2)^2 + (1-x(1))^2);
opts = optimoptions(@fmincon,'Algorithm','interior-point');
problem = createOptimProblem('fmincon','x0',randn(2,1),...
    'objective',anonrosen,'lb',[-2;-2],'ub',[2;2],...
    'options',opts);
```

Alternatives

You can create a problem structure by exporting from the Optimization app (`optimtool`), as described in “Exporting from the Optimization app” on page 3-9.

See Also

`optimtool` | `MultiStart` | `GlobalSearch`

CustomStartPointSet class

User-supplied start points

Description

An object wrapper of a matrix whose rows represent start points for `MultiStart`.

Construction

`*tpoints* = CustomStartPointSet(*ptmatrix*)` generates a `CustomStartPointSet` object from the *`ptmatrix`* matrix. Each row of *`ptmatrix`* represents one start point.

Properties

`DimStartPoints`

Dimension of each start point, a read-only property. `DimStartPoints` is the number of columns in *`ptmatrix`*.

`DimStartPoints` should be the same as the number of elements in `problem.x0`, the `problem` structure you pass to `run`.

`NumStartPoints`

Number of start points, a read-only property. This is the number of rows in *`ptmatrix`*.

Methods

list	List custom start points in set

Copy Semantics

Value. To learn how value classes affect copy operations, see Copying Objects in the MATLAB Programming Fundamentals documentation.

Examples

Create a `CustomStartPointSet` object with 40 three-dimensional rows. Each row represents a normally distributed random variable with mean `[10,10,10]` and variance `diag([4,4,4])`:

```
fortypts = 10*ones(40,3) + 4*randn(40,3); % a matrix
startpts = CustomStartPointSet(fortypts);
```

`startpts` is the `fortypts` matrix in an object wrapper.

Get the `fortypts` matrix from the `startpts` object of the previous example:

```
fortypts = list(startpts);
```

See Also

`list` | `RandomStartPointSet` | `MultiStart`

How To

- Class Attributes
- Property Attributes

ga

Find minimum of function using genetic algorithm

Syntax

```
x = ga(fitnessfcn,nvars)
x = ga(fitnessfcn,nvars,A,b)
x = ga(fitnessfcn,nvars,A,b,Aeq,beq)
x = ga(fitnessfcn,nvars,A,b,Aeq,beq,LB,UB)
x = ga(fitnessfcn,nvars,A,b,Aeq,beq,LB,UB,nonlcon)
x = ga(fitnessfcn,nvars,A,b,Aeq,beq,LB,UB,nonlcon,options)
x = ga(fitnessfcn,nvars,A,b,[],[],LB,UB,nonlcon,IntCon)
x = ga(fitnessfcn,nvars,A,b,[],[],LB,UB,nonlcon,IntCon,options)
x = ga(problem)
[x,fval] = ga(fitnessfcn,nvars,...)
[x,fval,exitflag] = ga(fitnessfcn,nvars,...)
[x,fval,exitflag,output] = ga(fitnessfcn,nvars,...)
[x,fval,exitflag,output,population] = ga(fitnessfcn,nvars,...)
[x,fval,exitflag,output,population,scores] = ga(fitnessfcn,
nvars,...)
```

Description

`x = ga(fitnessfcn,nvars)` finds a local unconstrained minimum, `x`, to the objective function, `fitnessfcn`. `nvars` is the dimension (number of design variables) of `fitnessfcn`.

`x = ga(fitnessfcn,nvars,A,b)` finds a local minimum `x` to `fitnessfcn`, subject to the linear inequalities `A*x` ≤ `b`. `ga` evaluates the matrix product `A*x` as if `x` is transposed (`A*x'`).

`x = ga(fitnessfcn,nvars,A,b,Aeq,beq)` finds a local minimum `x` to `fitnessfcn`, subject to the linear equalities `Aeq*x` = `beq` as well as `A*x` ≤ `b`. (Set `A=[]` and `b=[]` if no linear inequalities exist.) `ga` evaluates the matrix product `Aeq*x` as if `x` is transposed (`Aeq*x'`).

`x = ga(fitnessfcn,nvars,A,b,Aeq,beq,LB,UB)` defines a set of lower and upper bounds on the design variables, `x`, so that a solution is found in the range `LB` ≤ `x` ≤ `UB`. (Set `Aeq=[]` and `beq=[]` if no linear equalities exist.)

`x = ga(fitnessfcn,nvars,A,b,Aeq,beq,LB,UB,nonlcon)` subjects the minimization to the constraints defined in `nonlcon`. The function `nonlcon` accepts `x` and returns vectors `C` and `Ceq`, representing the nonlinear inequalities and equalities respectively. `ga` minimizes the `fitnessfcn` such that `C(x)` ≤ `0` and `Ceq(x) = 0`. (Set `LB=[]` and `UB=[]` if no bounds exist.)

`x = ga(fitnessfcn,nvars,A,b,Aeq,beq,LB,UB,nonlcon,options)` minimizes with the default optimization parameters replaced by values in the structure `options`, which can be created using the `gaoptimset` function. (Set `nonlcon=[]` if no nonlinear constraints exist.)

`x = ga(fitnessfcn,nvars,A,b,[],[],LB,UB,nonlcon,IntCon)` requires that the variables listed in `IntCon` take integer values.

Note: When there are integer constraints, `ga` does not accept linear or nonlinear equality constraints, only inequality constraints.

`x = ga(fitnessfcn,nvars,A,b,[],[],LB,UB,nonlcon,IntCon,options)` minimizes with integer constraints and with the default optimization parameters replaced by values in the `options` structure.

`x = ga(problem)` finds the minimum for `problem`, where `problem` is a structure.

`[x,fval] = ga(fitnessfcn,nvars,...)` returns `fval`, the value of the fitness function at `x`.

`[x,fval,exitflag] = ga(fitnessfcn,nvars,...)` returns `exitflag`, an integer identifying the reason the algorithm terminated.

`[x,fval,exitflag,output] = ga(fitnessfcn,nvars,...)` returns `output`, a structure that contains output from each generation and other information about the performance of the algorithm.

`[x,fval,exitflag,output,population] = ga(fitnessfcn,nvars,...)` returns the matrix, `population`, whose rows are the final population.

`[x,fval,exitflag,output,population,scores] = ga(fitnessfcn, nvars,...)` returns `scores` the scores of the final population.

Input Arguments

`fitnessfcn`

Handle to the fitness function. The fitness function should accept a row vector of length nvars and return a scalar value.

When the `'Vectorized'` option is `'on'`, `fitnessfcn` should accept a `pop`-by-`nvars` matrix, where `pop` is the current population size. In this case `fitnessfcn` should return a vector the same length as `pop` containing the fitness function values. `fitnessfcn` should not assume any particular size for `pop`, since `ga` can pass a single member of a population even in a vectorized calculation.

`nvars`

Positive integer representing the number of variables in the problem.

`A`

Matrix for linear inequality constraints of the form
`A*x` ≤ `b`.

If the problem has `m` linear inequality constraints and `nvars` variables, then

- `A` is a matrix of size `m`-by-`nvars`.
- `b` is a vector of length `m`.

`ga` evaluates the matrix product `A*x` as if `x` is transposed (`A*x'`).

Note: `ga` does not enforce linear constraints to be satisfied when the `PopulationType` option is `'bitString'` or `'custom'`.

`b`

Vector for linear inequality constraints of the form

A*x ≤ b.

If the problem has m linear inequality constraints and nvars variables, then

- A is a matrix of size m-by-nvars.
- b is a vector of length m.

Aeq

Matrix for linear equality constraints of the form
Aeq*x = beq.

If the problem has m linear equality constraints and nvars variables, then

- Aeq is a matrix of size m-by-nvars.
- beq is a vector of length m.

ga evaluates the matrix product Aeq*x as if x is transposed (Aeq*x').

Note: ga does not enforce linear constraints to be satisfied when the PopulationType option is 'bitString' or 'custom'.

beq

Vector for linear equality constraints of the form
Aeq*x = beq.

If the problem has m linear equality constraints and nvars variables, then

- Aeq is a matrix of size m-by-nvars.
- beq is a vector of length m.

LB

Vector of lower bounds. ga enforces that iterations stay above LB. Set LB(i) = –Inf if x(i)is unbounded below.

Note: ga does not enforce bounds to be satisfied when the PopulationType option is 'bitString' or 'custom'.

UB

Vector of upper bounds. `ga` enforces that iterations stay below `UB`. Set `UB(i) = Inf` if `x(i)` is unbounded above.

nonlcon

Function handle that returns two outputs:

```
[c,ceq] = nonlcon(x)
```

`ga` attempts to achieve `c` ≤ `0` and `ceq` = `0`. `c` and `ceq` are row vectors when there are multiple constraints. Set unused outputs to `[]`.

You can write `nonlcon` as a function handle to a file, such as

```
nonlcon = @constraintfile
```

where `constraintfile.m` is a file on your MATLAB path.

To learn how to use vectorized constraints, see “Vectorized Constraints” on page 2-7.

Note: `ga` does not enforce nonlinear constraints to be satisfied when the `PopulationType` option is set to `'bitString'` or `'custom'`.

If IntCon is not empty, the second output of `nonlcon` (`ceq`) must be empty (`[]`).

For information on how `ga` uses `nonlcon`, see “Nonlinear Constraint Solver Algorithms” on page 5-49.

---

options

Structure containing optimization options. Create `options` using `gaoptimset`, or by exporting options from the Optimization app as described in “Importing and Exporting Your Work” in the Optimization Toolbox documentation.

IntCon

Vector of positive integers taking values from `1` to nvars. Each value in `IntCon` represents an `x` component that is integer-valued.

Note: When `IntCon` is nonempty, Aeq and beq must be empty (`[]`), and nonlcon must return empty for `ceq`. For more information on integer programming, see “Mixed Integer Optimization” on page 5-27.

`problem`

Structure containing the following fields:

`fitnessfcn`	Fitness function
`nvars`	Number of design variables
`Aineq`	`A` matrix for linear inequality constraints
`Bineq`	`b` vector for linear inequality constraints
`Aeq`	`Aeq` matrix for linear equality constraints
`Beq`	`beq` vector for linear equality constraints
`lb`	Lower bound on `x`
`ub`	Upper bound on `x`
`nonlcon`	Nonlinear constraint function
`rngstate`	Optional field to reset the state of the random number generator
`intcon`	Index vector of integer variables
`solver`	`'ga'`
`options`	Options structure created using `gaoptimset` or the Optimization app

Create `problem` by exporting a problem from the Optimization app, as described in “Importing and Exporting Your Work” in the Optimization Toolbox documentation.

Output Arguments

`x`

Best point that `ga` located during its iterations.

`fval`

Fitness function evaluated at x.

`exitflag`

Integer giving the reason `ga` stopped iterating:

Exit Flag	Meaning
1	**Without nonlinear constraints** — Average cumulative change in value of the fitness function over `StallGenLimit` generations is less than `TolFun`, and the constraint violation is less than `TolCon`.
	With nonlinear constraints — Magnitude of the complementarity measure (see “Complementarity Measure” on page 11-16) is less than `sqrt(TolCon)`, the subproblem is solved using a tolerance less than `TolFun`, and the constraint violation is less than `TolCon`.
2	Fitness limit reached and the constraint violation is less than `TolCon`.
3	Value of the fitness function did not change in `StallGenLimit` generations and the constraint violation is less than `TolCon`.
4	Magnitude of step smaller than machine precision and the constraint violation is less than `TolCon`.
5	Minimum fitness limit `FitnessLimit` reached and the constraint violation is less than `TolCon`.
0	Maximum number of generations `Generations` exceeded.
-1	Optimization terminated by an output function or plot function.
-2	No feasible point found.
-4	Stall time limit `StallTimeLimit` exceeded.
-5	Time limit `TimeLimit` exceeded.

When there are integer constraints, `ga` uses the penalty fitness value instead of the fitness value for stopping criteria.

`output`

Structure containing output from each generation and other information about algorithm performance. The `output` structure contains the following fields:

- `problemtype` — String describing the type of problem, one of:

  - `'unconstrained'`
  - `'boundconstraints'`
  - `'linearconstraints'`
  - `'nonlinearconstr'`
  - `'integerconstraints'`
- `rngstate` — State of the MATLAB random number generator, just before the algorithm started. You can use the values in `rngstate` to reproduce the output of `ga`. See “Reproduce Results” on page 5-68.
- `generations` — Number of generations computed.
- `funccount` — Number of evaluations of the fitness function.
- `message` — Reason the algorithm terminated.
- `maxconstraint` — Maximum constraint violation, if any.

population

Matrix whose rows contain the members of the final population.

scores

Column vector of the fitness values (scores for `integerconstraints` problems) of the final population.

Examples

Given the following inequality constraints and lower bounds

$$\begin{bmatrix} 1 & 1 \\ -1 & 2 \\ 2 & 1 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \end{bmatrix} \le \begin{bmatrix} 2 \\ 2 \\ 3 \end{bmatrix},$$
$$x_1 \ge 0, \quad x_2 \ge 0,$$

use this code to find the minimum of the `lincontest6` function, which is provided in your software:

```
A = [1 1; -1 2; 2 1];
b = [2; 2; 3];
lb = zeros(2,1);
```

```
[x,fval,exitflag] = ga(@lincontest6,...
    2,A,b,[],[],lb)

Optimization terminated: average change in
the fitness value less than options.TolFun.

x =
    0.6700    1.3310

fval =
   -8.2218

exitflag =
     1
```

Optimize a function where some variables must be integers:

```
fun = @(x) (x(1) - 0.2)^2 + ...
    (x(2) - 1.7)^2 + (x(3) - 5.1)^2;
x = ga(fun,3,[],[],[],[],[],[],[], ...
   [2 3]) % variables 2 and 3 are integers

Optimization terminated: average change
in the penalty fitness value less
than options.TolFun and constraint violation
is less than options.TolCon.

x =
    0.2000    2.0000    5.0000
```

Alternatives

For problems without integer constraints, consider using `patternsearch` instead of `ga`.

More About

Complementarity Measure

In the nonlinear constraint solver, the complementarity measure is the norm of the vector whose elements are $c_i\lambda_i$, where c_i is the nonlinear inequality constraint violation, and λ_i is the corresponding Lagrange multiplier.

Tips

- To write a function with additional parameters to the independent variables that can be called by `ga`, see "Passing Extra Parameters" in the Optimization Toolbox documentation.
- For problems that use the population type `Double Vector` (the default), `ga` does not accept functions whose inputs are of type `complex`. To solve problems involving complex data, write your functions so that they accept real vectors, by separating the real and imaginary parts.

Algorithms

For a description of the genetic algorithm, see "How the Genetic Algorithm Works" on page 5-18.

For a description of the mixed integer programming algorithm, see "Integer ga Algorithm" on page 5-35.

For a description of the nonlinear constraint algorithm, see "Nonlinear Constraint Solver Algorithms" on page 5-49.

- "Genetic Algorithm"
- "Getting Started with Global Optimization Toolbox"
- "Optimization Problem Setup"

References

[1] Goldberg, David E., *Genetic Algorithms in Search, Optimization & Machine Learning*, Addison-Wesley, 1989.

[2] A. R. Conn, N. I. M. Gould, and Ph. L. Toint. "A Globally Convergent Augmented Lagrangian Algorithm for Optimization with General Constraints and Simple Bounds", *SIAM Journal on Numerical Analysis*, Volume 28, Number 2, pages 545–572, 1991.

[3] A. R. Conn, N. I. M. Gould, and Ph. L. Toint. "A Globally Convergent Augmented Lagrangian Barrier Algorithm for Optimization with General Inequality Constraints and Simple Bounds", *Mathematics of Computation*, Volume 66, Number 217, pages 261–288, 1997.

See Also

`gamultiobj` | `gaoptimset` | `particleswarm` | `patternsearch`

gamultiobj

Find minima of multiple functions using genetic algorithm

Syntax

```
X = gamultiobj(FITNESSFCN,NVARS)
X = gamultiobj(FITNESSFCN,NVARS,A,b)
X = gamultiobj(FITNESSFCN,NVARS,A,b,Aeq,beq)
X = gamultiobj(FITNESSFCN,NVARS,A,b,Aeq,beq,LB,UB)
X = gamultiobj(FITNESSFCN,NVARS,A,b,Aeq,beq,LB,UB,nonlcon)
X = gamultiobj(FITNESSFCN,NVARS,A,b,Aeq,beq,LB,UB,options)
X = gamultiobj(FITNESSFCN,NVARS,A,b,Aeq,beq,LB,UB,nonlcon,options)
X = gamultiobj(problem)
[X,FVAL] = gamultiobj(FITNESSFCN,NVARS, ...)
[X,FVAL,EXITFLAG] = gamultiobj(FITNESSFCN,NVARS, ...)
[X,FVAL,EXITFLAG,OUTPUT] = gamultiobj(FITNESSFCN,NVARS, ...)
[X,FVAL,EXITFLAG,OUTPUT,POPULATION] = gamultiobj(FITNESSFCN, ...)
[X,FVAL,EXITFLAG,OUTPUT,POPULATION,SCORE] =
gamultiobj(FITNESSFCN, ...)
```

Description

`gamultiobj` implements the genetic algorithm at the command line to minimize a multicomponent objective function.

`X = gamultiobj(FITNESSFCN,NVARS)` finds a local Pareto set `X` of the objective functions defined in `FITNESSFCN`. For details on writing `FITNESSFCN`, see "Compute Objective Functions" on page 2-2. `NVARS` is the dimension of the optimization problem (number of decision variables). `X` is a matrix with `NVARS` columns. The number of rows in `X` is the same as the number of Pareto solutions. All solutions in a Pareto set are equally optimal; it is up to the designer to select a solution in the Pareto set depending on the application.

`X = gamultiobj(FITNESSFCN,NVARS,A,b)` finds a local Pareto set `X` of the objective functions defined in `FITNESSFCN`, subject to the linear inequalities $A * x \le b$, see "Linear Inequality Constraints". Linear constraints are supported only for the default `PopulationType` option (`'doubleVector'`). Other population types, e.g., `'bitString'` and `'custom'`, are not supported.

`X = gamultiobj(FITNESSFCN,NVARS,A,b,Aeq,beq)` finds a local Pareto set `X` of the objective functions defined in `FITNESSFCN`, subject to the linear equalities $Aeq * x = beq$ as well as the linear inequalities $A * x \leq b$, see "Linear Equality Constraints". (Set `A=[]` and `b=[]` if no inequalities exist.) Linear constraints are supported only for the default `PopulationType` option (`'doubleVector'`). Other population types, e.g., `'bitString'` and `'custom'`, are not supported.

`X = gamultiobj(FITNESSFCN,NVARS,A,b,Aeq,beq,LB,UB)` defines a set of lower and upper bounds on the design variables `X` so that a local Pareto set is found in the range $LB \leq x \leq UB$, see "Bound Constraints". Use empty matrices for `LB` and `UB` if no bounds exist. Bound constraints are supported only for the default `PopulationType` option (`'doubleVector'`). Other population types, e.g., `'bitString'` and `'custom'`, are not supported.

`X = gamultiobj(FITNESSFCN,NVARS,A,b,Aeq,beq,LB,UB,nonlcon)` subjects the minimization to the constraints defined in `nonlcon`. The function `nonlcon` accepts `x` and returns vectors `C` and `Ceq`, representing the nonlinear inequalities and equalities respectively. `gamultiobj` minimizes the `fitnessfcn` such that `C(x)` ≤ `0` and `Ceq(x) = 0`. (Set `LB=[]` and `UB=[]` if no bounds exist.)

`X = gamultiobj(FITNESSFCN,NVARS,A,b,Aeq,beq,LB,UB,options)` or `X = gamultiobj(FITNESSFCN,NVARS,A,b,Aeq,beq,LB,UB,nonlcon,options)` finds a Pareto set `X` with the default optimization parameters replaced by values in the structure `options`. `options` can be created with the `gaoptimset` function.

`X = gamultiobj(problem)` finds the Pareto set for `problem`, where `problem` is a structure containing the following fields:

`fitnessfcn`	Fitness functions
`nvars`	Number of design variables
`Aineq`	`A` matrix for linear inequality constraints
`bineq`	`b` vector for linear inequality constraints
`Aeq`	`Aeq` matrix for linear equality constraints
`beq`	`beq` vector for linear equality constraints

lb	Lower bound on x
ub	Upper bound on x
nonlcon	Nonlinear constraint function (optional)
solver	'gamultiobj'
rngstate	Optional field to reset the state of the random number generator
options	Options structure created using gaoptimset

Create the structure `problem` by exporting a problem from Optimization app, as described in "Importing and Exporting Your Work" in the Optimization Toolbox documentation.

`[X,FVAL] = gamultiobj(FITNESSFCN,NVARS, ...)` returns a matrix `FVAL`, the value of all the objective functions defined in `FITNESSFCN` at all the solutions in `X`. `FVAL` has `numberOfObjectives` columns and same number of rows as does `X`.

`[X,FVAL,EXITFLAG] = gamultiobj(FITNESSFCN,NVARS, ...)` returns `EXITFLAG`, which describes the exit condition of `gamultiobj`. Possible values of `EXITFLAG` and the corresponding exit conditions are listed in this table.

EXITFLAG Value	Exit Condition
1	Average change in value of the spread over `options.StallGenLimit` generations less than `options.TolFun`, and the final spread is less than the average spread over the past `options.StallGenLimit` generations
0	Maximum number of generations exceeded
-1	Optimization terminated by an output function or plot function
-2	No feasible point found
-5	Time limit exceeded

`[X,FVAL,EXITFLAG,OUTPUT] = gamultiobj(FITNESSFCN,NVARS, ...)` returns a structure `OUTPUT` with the following fields:

OUTPUT Field	Meaning
problemtype	Type of problem:

OUTPUT Field	Meaning
	• `'unconstrained'` — No constraints • `'boundconstraints'` — Only bound constraints • `'linearconstraints'` — Linear constraints, with or without bound constraints
`rngstate`	State of the MATLAB random number generator, just before the algorithm started. You can use the values in `rngstate` to reproduce the output of `ga`. See "Reproduce Results" on page 5-68.
`generations`	Total number of generations, excluding `HybridFcn` iterations
`funccount`	Total number of function evaluations
`message`	`gamultiobj` termination message
`averagedistance`	Average "distance," which by default is the standard deviation of the norm of the difference between Pareto front members and their mean
`spread`	Combination of the "distance," and a measure of the movement of the points on the Pareto front between the final two iterations
`maxconstraint`	Maximum constraint violation at the final Pareto set

`[X,FVAL,EXITFLAG,OUTPUT,POPULATION] = gamultiobj(FITNESSFCN, ...)` returns the final `POPULATION` at termination.

`[X,FVAL,EXITFLAG,OUTPUT,POPULATION,SCORE] = gamultiobj(FITNESSFCN, ...)` returns the `SCORE` of the final `POPULATION`.

Examples

This example optimizes two objectives defined by Schaffer's second function, which has two objectives and a scalar input argument. The Pareto front is disconnected. Define this function in a file:

```
function y = schaffer2(x) % y has two columns

% Initialize y for two objectives and for all x
y = zeros(length(x),2); % ready for vectorization
```

```
% Evaluate first objective.
% This objective is piecewise continuous.
for i = 1:length(x)
    if x(i) <= 1
        y(i,1) = -x(i);
    elseif x(i) <=3
        y(i,1) = x(i) -2;
    elseif x(i) <=4
        y(i,1) = 4 - x(i);
    else
        y(i,1) = x(i) - 4;
    end
end

% Evaluate second objective
y(:,2) = (x -5).^2;
```

First, plot the two objectives:

```
x = -1:0.1:8;
y = schaffer2(x);

plot(x,y(:,1),'.r'); hold on
plot(x,y(:,2),'.b');
```

The two component functions compete in the range [1, 3] and [4, 5]. But the Pareto-optimal front consists of only two disconnected regions: [1, 2] and [4, 5]. This is because the region [2, 3] is inferior to [1, 2].

Next, impose a bound constraint on `x`, $-5 \le x \le 10$ setting

```
lb = -5;
ub = 10;
```

The best way to view the results of the genetic algorithm is to visualize the Pareto front directly using the `@gaplotpareto` option. To optimize Schaffer's function, a larger population size than the default (15) is needed, because of the disconnected front. This example uses 60. Set the optimization options as:

```
options = gaoptimset('PopulationSize',60,'PlotFcns',@gaplotpareto);
```

Now call `gamultiobj`, specifying one independent variable and only the bound constraints:

```
[x,f,exitflag] = gamultiobj(@schaffer2,1,[],[],[],[],...
    lb,ub,options);

Optimization terminated: average change in the spread of
Pareto solutions less than options.TolFun.

exitflag
exitflag = 1
```

The vectors `x`, `f(:,1)`, and `f(:,2)` respectively contain the Pareto set and both objectives evaluated on the Pareto set.

Examples Included in the Toolbox

The gamultiobjfitness example solves a simple problem with one decision variable and two objectives.

The `gamultiobjoptionsdemo` example shows how to set options for multiobjective optimization.

More About

Algorithms

`gamultiobj` uses a controlled elitist genetic algorithm (a variant of NSGA-II [1]). An elitist GA always favors individuals with better fitness value (rank). A controlled elitist GA also favors individuals that can help increase the diversity of the population even if they have a lower fitness value. It is important to maintain the diversity of population for convergence to an optimal Pareto front. Diversity is maintained by controlling the elite members of the population as the algorithm progresses. Two options, `ParetoFraction` and `DistanceFcn`, control the elitism. `ParetoFraction` limits the number of individuals on the Pareto front (elite members). The distance function, selected by `DistanceFcn`, helps to maintain diversity on a front by favoring individuals that are relatively far away on the front. The algorithm stops if the *spread*, a measure of the movement of the Pareto front, is small.

References

[1] Deb, Kalyanmoy. *Multi-Objective Optimization Using Evolutionary Algorithms*. John Wiley & Sons, 2001.

See Also

`ga` | `gaoptimget` | `gaoptimset` | `patternsearch` | `Special Characters` | `rand` | `randn`

gaoptimget

Obtain values of genetic algorithm options structure

Syntax

```
val = gaoptimget(options, 'name')
val = gaoptimget(options, 'name', default)
```

Description

`val = gaoptimget(options, 'name')` returns the value of the parameter `name` from the genetic algorithm options structure `options`. `gaoptimget(options, 'name')` returns an empty matrix `[]` if the value of `name` is not specified in `options`. It is only necessary to type enough leading characters of `name` to uniquely identify it. `gaoptimget` ignores case in parameter names.

`val = gaoptimget(options, 'name', default)` returns the `'name'` parameter, but will return the default value if the name parameter is not specified (or is `[]`) in `options`.

More About

- "Genetic Algorithm Options" on page 10-28

See Also

`ga` | `gamultiobj` | `gaoptimset`

gaoptimset

Create genetic algorithm options structure

Syntax

```
gaoptimset
options = gaoptimset
options = gaoptimset(@ga)
options = gaoptimset(@gamultiobj)
options = gaoptimset('param1',value1,'param2',value2,...)
options = gaoptimset(oldopts,'param1',value1,...)
options = gaoptimset(oldopts,newopts)
```

Description

`gaoptimset` with no input or output arguments displays a complete list of parameters with their valid values.

`options = gaoptimset` (with no input arguments) creates a structure called `options` that contains the options, or *parameters*, for the genetic algorithm and sets parameters to `[]`, indicating default values will be used.

`options = gaoptimset(@ga)` creates a structure called `options` that contains the default options for the genetic algorithm.

`options = gaoptimset(@gamultiobj)` creates a structure called `options` that contains the default options for `gamultiobj`.

`options = gaoptimset('param1',value1,'param2',value2,...)` creates a structure called `options` and sets the value of `'param1'` to `value1`, `'param2'` to `value2`, and so on. Any unspecified parameters are set to their default values. It is sufficient to type only enough leading characters to define the parameter name uniquely. Case is ignored for parameter names.

`options = gaoptimset(oldopts,'param1',value1,...)` creates a copy of `oldopts`, modifying the specified parameters with the specified values.

`options = gaoptimset(oldopts,newopts)` combines an existing options structure, `oldopts`, with a new options structure, `newopts`. Any parameters in `newopts` with nonempty values overwrite the corresponding old parameters in `oldopts`.

Options

The following table lists the options you can set with `gaoptimset`. See "Genetic Algorithm Options" on page 10-28 for a complete description of these options and their values. Values in `{}` denote the default value. `{}*` means the default when there are linear constraints, and for `MutationFcn` also when there are bounds. You can also view the optimization parameters and defaults by typing `gaoptimset` at the command line. **I*** indicates that `ga` ignores or overwrites the option for mixed integer optimization problems.

Option	Description	Values
`CreationFcn`	**I*** Handle to the function that creates the initial population. See "Population Options" on page 10-32.	`{@gacreationuniform}` | `{@gacreationlinearfeasible}*`
`CrossoverFcn`	**I*** Handle to the function that the algorithm uses to create crossover children. See "Crossover Options" on page 10-42.	`@crossoverheuristic` | `{@crossoverscattered}` | `{@crossoverintermediate}*` | `@crossoversinglepoint` | `@crossovertwopoint` | `@crossoverarithmetic`
`CrossoverFraction`	The fraction of the population at the next generation, not including elite children, that is created by the crossover function	Positive scalar | `{0.8}`
`Display`	Level of display	`'off'` | `'iter'` | `'diagnose'` | `{'final'}`
`DistanceMeasureFcn`	**I*** Handle to the function that computes distance measure of individuals, computed in decision variable or design	`{@distancecrowding,'phenotype'}`

Option	Description	Values
	space (genotype) or in function space (phenotype)	
`EliteCount`	Positive integer specifying how many individuals in the current generation are guaranteed to survive to the next generation. Not used in `gamultiobj`.	Positive integer | `{ceil(0.05*PopulationSize)}` | `{0.05*(default PopulationSize)}` for mixed-integer problems
`FitnessLimit`	Scalar. If the fitness function attains the value of `FitnessLimit`, the algorithm halts.	Scalar | `{-Inf}`
`FitnessScalingFcn`	Handle to the function that scales the values of the fitness function	`@fitscalingshiftlinear` | `@fitscalingprop` | `@fitscalingtop` | `{@fitscalingrank}`
`Generations`	Positive integer specifying the maximum number of iterations before the algorithm halts	Positive integer |`{100*numberOfVariables}`
`HybridFcn`	**I*** Handle to a function that continues the optimization after `ga` terminates or Cell array specifying the hybrid function and its options structure	Function handle | `@fminsearch` | `@patternsearch` | `@fminunc` | `@fmincon` | `{[]}` or 1-by-2 cell array | `{@solver, hybridoptions}`, where `solver = fminsearch`, `patternsearch`, `fminunc`, or `fmincon` `{[]}`
`InitialPenalty`	**I*** Initial value of penalty parameter	Positive scalar | `{10}`
`InitialPopulation`	Initial population used to seed the genetic algorithm; can be partial	Matrix | `{[]}`
`InitialScores`	**I*** Initial scores used to determine fitness; can be partial	Column vector | `{[]}`

Option	Description	Values
`MigrationDirection`	Direction of migration — see "Migration Options" on page 10-46	`'both'` | `{'forward'}`
`MigrationFraction`	Scalar between 0 and 1 specifying the fraction of individuals in each subpopulation that migrates to a different subpopulation — see "Migration Options" on page 10-46	Scalar | `{0.2}`
`MigrationInterval`	Positive integer specifying the number of generations that take place between migrations of individuals between subpopulations — see "Migration Options" on page 10-46	Positive integer | `{20}`
`MutationFcn`	**I*** Handle to the function that produces mutation children. See "Mutation Options" on page 10-39.	`@mutationuniform` | `{@mutationadaptfeasible}*` | `{@mutationgaussian}`
`NonlinConAlgorithm`	Nonlinear constraint algorithm. See "Nonlinear Constraint Solver Algorithms" on page 5-49.	`{'auglag'}` | `'penalty'`
`OutputFcns`	Functions that `ga` calls at each iteration. See "Output Function Options" on page 10-50.	Function handle or cell array of function handles | `{[]}`
`ParetoFraction`	**I*** Scalar between 0 and 1 specifying the fraction of individuals to keep on the first Pareto front while the solver selects individuals from higher fronts	Scalar | `{0.35}`
`PenaltyFactor`	**I*** Penalty update parameter	Positive scalar | `{100}`

Option	Description	Values
`PlotFcns`	Array of handles to functions that plot data computed by the algorithm. See "Plot Options" on page 10-29.	`@gaplotbestf | @gaplotbestindiv | @gaplotdistance | @gaplotexpectation | @gaplotgenealogy | @gaplotmaxconstr | @gaplotrange | @gaplotselection | @gaplotscorediversity | @gaplotscores | @gaplotstopping | {[]}` For `gamultiobj` there are additional choices: `@gaplotpareto | @gaplotparetodistance | @gaplotrankhist | @gaplotspread`
`PlotInterval`	Positive integer specifying the number of generations between consecutive calls to the plot functions	Positive integer | `{1}`
`PopInitRange`	Matrix or vector specifying the range of the individuals in the initial population. Applies to `gacreationuniform` creation function. `ga` shifts and scales the default initial range to match any finite bounds.	Matrix or vector | `{[-10;10]}` for unbounded components, `{[-1e4+1;1e4+1]}` for unbounded components of integer-constrained problems, `{[lb;ub]}` for bounded components, with the default range modified to match one-sided bounds.
`PopulationSize`	Size of the population	Positive integer | `{50}` when `numberOfVariables <= 5`, `{200}` otherwise | `{min(max(10*nvars,40),100)}` for mixed-integer problems

Option	Description	Values
`PopulationType`	String describing the data type of the population — must be `'doubleVector'` for mixed integer problems	`'bitstring'` | `'custom'` | `{'doubleVector'}` `ga` ignores all constraints when `PopulationType` is set to `'bitString'` or `'custom'`. See “Population Options” on page 10-32.
`SelectionFcn`	**I*** Handle to the function that selects parents of crossover and mutation children	`@selectionremainder` | `@selectionuniform` | `{@selectionstochunif}` | `@selectionroulette` | `@selectiontournament`
`StallGenLimit`	Positive integer. The algorithm stops if the average relative change in the best fitness function value over `StallGenLimit` generations is less than or equal to `TolFun`. If `StallTest` is `'geometricWeighted'`, then the algorithm stops if the *weighted* average relative change is less than or equal to `TolFun`.	Positive integer | `{50}`
`StallTest`	String describing the stopping test.	`'geometricWeighted'` | `{'totalChange'}`
`StallTimeLimit`	Positive scalar. The algorithm stops if there is no improvement in the objective function for `StallTimeLimit` seconds, as measured by `cputime`.	Positive scalar | `{Inf}`
`TimeLimit`	Positive scalar. The algorithm stops after running for `TimeLimit` seconds, as measured by `cputime`.	Positive scalar | `{Inf}`

Option	Description	Values
`TolCon`	Positive scalar. `TolCon` is used to determine the feasibility with respect to nonlinear constraints. Also, `max(sqrt(eps),sqrt(TolCon` determines feasibility with respect to linear constraints.	Positive scalar | `{1e-6}`
`TolFun`	Positive scalar. The algorithm stops if the average relative change in the best fitness function value over `StallGenLimit` generations is less than or equal to `TolFun`. If `StallTest` is `'geometricWeighted'`, then the algorithm stops if the *weighted* average relative change is less than or equal to `TolFun`.	Positive scalar | `{1e-6}`
`UseParallel`	Compute fitness and nonlinear constraint functions in parallel, see "Vectorize and Parallel Options (User Function Evaluation)" on page 10-52 and "How to Use Parallel Processing" on page 9-12.	`true` | `{false}`
`Vectorized`	Specifies whether functions are vectorized, see "Vectorize and Parallel Options (User Function Evaluation)" on page 10-52 and "Vectorize the Fitness Function" on page 5-110.	`'on'` | `{'off'}`

More About

- “Genetic Algorithm Options” on page 10-28

See Also

`ga` | `gamultiobj` | `gaoptimget`

GlobalOptimSolution class

Optimization solution

Description

Information on a local minimum, including location, objective function value, and start point or points that lead to the minimum.

`GlobalSearch` and `MultiStart` generate a vector of `GlobalOptimSolution` objects. The vector is ordered by objective function value, from lowest (best) to highest (worst).

Construction

When you run them, `GlobalSearch` and `MultiStart` create `GlobalOptimSolution` objects as output.

Properties

`Exitflag`

An integer describing the result of the local solver run.

For the meaning of the exit flag, see the description in the appropriate local solver function reference page:

- `fmincon`
- `fminunc`
- `lsqcurvefit`
- `lsqnonlin`

`Fval`

Objective function value at the solution.

Output

Output structure returned by the local solver.

X

Solution point, with the same dimensions as the initial point.

X0

Cell array of start points that led to the solution.

Copy Semantics

Value. To learn how value classes affect copy operations, see Copying Objects in the MATLAB Programming Fundamentals documentation.

Examples

Use `MultiStart` to create a vector of `GlobalOptimSolution` objects:

```
ms = MultiStart;
sixmin = @(x)(4*x(1)^2 - 2.1*x(1)^4 + x(1)^6/3 ...
    + x(1)*x(2) - 4*x(2)^2 + 4*x(2)^4);
problem = createOptimProblem('fmincon','x0',[-1,2],...
    'objective',sixmin,'lb',[-3,-3],'ub',[3,3]);
[xmin,fmin,flag,outpt,allmins] = run(ms,problem,30);
```

`allmins` is the vector of `GlobalOptimSolution` objects:

```
allmins

allmins =

  1x30 GlobalOptimSolution

  Properties:
    X
    Fval
    Exitflag
    Output
```

```
X0
```

See Also

`MultiStart.run` | `GlobalSearch` | `MultiStart` | `GlobalSearch.run`

How To

- Class Attributes
- Property Attributes

GlobalSearch class

Find global minimum

Description

A `GlobalSearch` object contains properties (options) that affect how the `run` method searches for a global minimum, or generates a `GlobalOptimSolution` object.

Construction

`gs` `= GlobalSearch` constructs a new global search optimization solver with its properties set to the defaults.

`gs` `= GlobalSearch('`*`PropertyName`*`',`*`PropertyValue`*`,...)` constructs the object using options, specified as property name and value pairs.

`gs` `= GlobalSearch(`*`oldgs`*`,'`*`PropertyName`*`',`*`PropertyValue`*`,...)` constructs a copy of the `GlobalSearch` solver `oldgs`. The *`gs`* object has the named properties altered with the specified values.

`gs` `= GlobalSearch(`*`ms`*`)` constructs *`gs`*, a `GlobalSearch` solver, with common parameter values from the *`ms`* MultiStart solver.

Properties

`BasinRadiusFactor`

A basin radius decreases after `MaxWaitCycle` consecutive start points are within the basin. The basin radius decreases by a factor of 1 – `BasinRadiusFactor`.

Set `BasinRadiusFactor` to `0` to disable updates of the basin radius.

Default: `0.2`

`Display`

Detail level of iterative display. Possible values:

- `'final'` — Report summary results after `run` finishes.
- `'iter'` — Report results after the initial `fmincon` run, after Stage 1, after every 200 start points, and after every run of `fmincon`, in addition to the final summary.
- `'off'` — No display.

Default: `'final'`

`DistanceThresholdFactor`

A multiplier for determining whether a trial point is in an existing basin of attraction. For details, see "Examine Stage 2 Trial Point to See if fmincon Runs" on page 3-48.

Default: `0.75`

`MaxTime`

Time in seconds for a run. `GlobalSearch` halts when it sees `MaxTime` seconds have passed since the beginning of the run.

Default: `Inf`

`MaxWaitCycle`

A positive integer tolerance appearing in several points in the algorithm:

- If the observed penalty function of `MaxWaitCycle` consecutive trial points is at least the penalty threshold, then raise the penalty threshold (see `PenaltyThresholdFactor`).
- If `MaxWaitCycle` consecutive trial points are in a basin, then update that basin's radius (see `BasinRadiusFactor`).

Default: `20`

`NumStageOnePoints`

Number of start points in Stage 1. For details, see "Obtain Stage 1 Start Point, Run" on page 3-47.

Default: `200`

`NumTrialPoints`

Number of potential start points to examine in addition to `x0` from the `problem` structure. `GlobalSearch` runs only those potential start points that pass several tests. For more information, see "GlobalSearch Algorithm" on page 3-46.

Default: `1000`

`OutputFcns`

A function handle or cell array of function handles to output functions. Output functions run after each local solver call. They also run when the global solver starts and ends. Write your output functions using the syntax described in "OutputFcns" on page 10-3. See "GlobalSearch Output Function" on page 3-37.

Default: `[]`

`PenaltyThresholdFactor`

Determines increase in the penalty threshold. For details, see React to Large Counter Values.

Default: `0.2`

`PlotFcns`

A function handle or cell array of function handles to plot functions. Plot functions run after each local solver call. They also run when the global solver starts and ends. Write your plot functions using the syntax described in "OutputFcns" on page 10-3. There are two built-in plot functions:

- `@gsplotbestf` plots the best objective function value.
- `@gsplotfunccount` plots the number of function evaluations.

See "MultiStart Plot Function" on page 3-41.

Default: `[]`

`StartPointsToRun`

Directs the solver to exclude certain start points from being run:

- `all` — Accept all start points.
- `bounds` — Reject start points that do not satisfy bounds.
- `bounds-ineqs` — Reject start points that do not satisfy bounds or inequality constraints.

`GlobalSearch` checks the `StartPointsToRun` property only during Stage 2 of the `GlobalSearch` algorithm (the main loop). For more information, see "GlobalSearch Algorithm" on page 3-46.

Default: `'all'`

`TolFun`

Describes how close two objective function values must be for solvers to consider them identical for creating the vector of local solutions. Solvers consider two solutions identical if they are within `TolX` distance of each other and have objective function values within `TolFun` of each other. If both conditions are not met, solvers report the solutions as distinct. Set `TolFun` to `0` to obtain the results of every local solver run. Set `TolFun` to a larger value to have fewer results.

Default: `1e-6`

`TolX`

Describes how close two points must be for solvers to consider them identical for creating the vector of local solutions. Solvers compute the distance between a pair of points with `norm`, the Euclidean distance. Solvers consider two solutions identical if they are within `TolX` distance of each other and have objective function values within `TolFun` of each other. If both conditions are not met, solvers report the solutions as distinct. Set `TolX` to `0` to obtain the results of every local solver run. Set `TolX` to a larger value to have fewer results.

Default: `1e-6`

Methods

run	Find global minimum

Copy Semantics

Value. To learn how value classes affect copy operations, see Copying Objects in the MATLAB Programming Fundamentals documentation.

Examples

Solve a problem using a default `GlobalSearch` object:

```
opts = optimoptions(@fmincon,'Algorithm','interior-point');
problem = createOptimProblem('fmincon','objective',...
 @(x) x.^2 + 4*sin(5*x),'x0',3,'lb',-5,'ub',5,'options',opts);
gs = GlobalSearch;
[x,f] = run(gs,problem)
```

Algorithms

A detailed description of the algorithm appears in “GlobalSearch Algorithm” on page 3-46. Ugray et al. [1] describes both the algorithm and the scatter-search method of generating trial points.

References

[1] Ugray, Zsolt, Leon Lasdon, John Plummer, Fred Glover, James Kelly, and Rafael Martí. *Scatter Search and Local NLP Solvers: A Multistart Framework for Global Optimization*. INFORMS Journal on Computing, Vol. 19, No. 3, 2007, pp. 328–340.

See Also

`GlobalOptimSolution` | `MultiStart` | `createOptimProblem`

list

Class: CustomStartPointSet

List custom start points in set

Syntax

`tpts = list(CS)`

Description

`tpts = list(CS)` returns the matrix of start points in the `CS` `CustomStartPoints` object.

Input Arguments

`CS`

A `CustomStartPointSet` object.

Output Arguments

`tpts`

Matrix of start points. The rows of `tpts` represent the start points.

Examples

Create a `CustomStartPointSet` containing 40 seven-dimensional normally distributed points, then use `list` to get the matrix of points from the object:

```
startpts = randn(40,7) % 40 seven-dimensional start points
cs = CustomStartPointSet(startpts); % cs is an object
```

```
startpts2 = list(cs) % startpts2 = startpts
```

See Also

`CustomStartPointSet` | `createOptimProblem`

list

Class: RandomStartPointSet

Generate start points

Syntax

`points = list(RandSet,problem)`

Description

`points` `= list(`*`RandSet`*`,`*`problem`*`)` generates pseudorandom start points using the parameters in the *`RandSet`* `RandomStartPointSet` object, and information from the *`problem`* problem structure.

Input Arguments

`RandSet`

A `RandomStartPointSet` object. This contains parameters for generating the points: number of points, and artificial bounds.

`problem`

An optimization problem structure. `list` generates points uniformly within the bounds of the `problem` structure. If a component is unbounded, `list` uses the artificial bounds from `RandSet`. `list` takes the dimension of the points from the `x0` field in `problem`.

Output Arguments

`points`

A `k`-by-`n` matrix. The number of rows `k` is the number of start points that `RandSet` specifies. The number of columns `n` is the dimension of the start points. `n` is equal to the number of elements in the `x0` field in `problem`. The `MultiStart` algorithm uses each row of `points` as an initial point in an optimization.

Examples

Create a matrix representing 40 seven-dimensional start points:

```
rs = RandomStartPointSet('NumStartPoints',40); % 40 points
problem = createOptimProblem('fminunc','x0',ones(7,1),...
    'objective',@rosenbrock);
ptmatrix = list(rs,problem); % matrix values between
   % -1000 and 1000 since those are the default bounds
   % for unconstrained dimensions
```

Algorithms

The `list` method generates a pseudorandom random matrix using the default random number stream. Each row of the matrix represents a start point to run. `list` generates points that satisfy the bounds in `problem`. If `lb` is the vector of lower bounds, `ub` is the vector of upper bounds, there are `n` dimensions in a point, and there are `k` rows in the matrix, the random matrix is

```
lb + (ub - lb).*rand(k,n)
```

- If a component has no bounds, `RandomStartPointSet` uses a lower bound of `-ArtificialBound`, and an upper bound of `ArtificialBound`.
- If a component has a lower bound `lb`, but no upper bound, `RandomStartPointSet` uses an upper bound of `lb + 2*ArtificialBound`.
- Similarly, if a component has an upper bound `ub`, but no lower bound, `RandomStartPointSet` uses a lower bound of `ub - 2*ArtificialBound`.

The default value of `ArtificialBound` is `1000`.

To obtain identical pseudorandom results, reset the default random number stream. See "Reproduce Results" on page 3-69.

See Also

`RandomStartPointSet` | `createOptimProblem` | `MultiStart`

MultiStart class

Find multiple local minima

Description

A `MultiStart` object contains properties (options) that affect how the `run` method repeatedly runs a local solver, or generates a `GlobalOptimSolution` object.

Construction

`MS` `= MultiStart` constructs *`MS`*, a `MultiStart` solver with its properties set to the defaults.

`MS` `= MultiStart('`*`PropertyName`*`',`*`PropertyValue`*`,...)` constructs *`MS`* using options, specified as property name and value pairs.

`MS` `= MultiStart(`*`oldMS`*`,'`*`PropertyName`*`',`*`PropertyValue`*`,...)` creates a copy of the *`oldMS`* `MultiStart` solver, with the named properties changed to the specified values.

`MS` `= MultiStart(`*`GS`*`)` constructs *`MS`*, a `MultiStart` solver, with common parameter values from the *`GS`* `GlobalSearch` solver.

Properties

`Display`

Detail level of the output to the Command Window:

- `'final'` — Report summary results after `run` finishes.
- `'iter'` — Report results after each local solver run, in addition to the final summary.
- `'off'` — No display.

Default: `final`

`MaxTime`

Tolerance on the time `MultiStart` runs. `MultiStart` and its local solvers halt when they see `MaxTime` seconds have passed since the beginning of the run. Time means *wall clock* as opposed to processor cycles.

Default: `Inf`

`OutputFcns`

A function handle or cell array of function handles to output functions. Output functions run after each local solver call. They also run when the global solver starts and ends. Write your output functions using the syntax described in "OutputFcns" on page 10-3. See "GlobalSearch Output Function" on page 3-37.

Default: `[]`

`PlotFcns`

A function handle or cell array of function handles to plot functions. Plot functions run after each local solver call. They also run when the global solver starts and ends. Write your plot functions using the syntax described in "OutputFcns" on page 10-3. There are two built-in plot functions:

- `@gsplotbestf` plots the best objective function value.
- `@gsplotfunccount` plots the number of function evaluations.

See "MultiStart Plot Function" on page 3-41.

Default: `[]`

`StartPointsToRun`

Directs the solver to exclude certain start points from being run:

- `all` — Accept all start points.
- `bounds` — Reject start points that do not satisfy bounds.
- `bounds-ineqs` — Reject start points that do not satisfy bounds or inequality constraints.

Default: `all`

`TolFun`

Describes how close two objective function values must be for solvers to consider them identical for creating the vector of local solutions. Solvers consider two solutions identical if they are within `TolX` distance of each other and have objective function values within `TolFun` of each other. If both conditions are not met, solvers report the solutions as distinct. Set `TolFun` to `0` to obtain the results of every local solver run. Set `TolFun` to a larger value to have fewer results.

Default: `1e-6`

`TolX`

Describes how close two points must be for solvers to consider them identical for creating the vector of local solutions. Solvers compute the distance between a pair of points with `norm`, the Euclidean distance. Solvers consider two solutions identical if they are within `TolX` distance of each other and have objective function values within `TolFun` of each other. If both conditions are not met, solvers report the solutions as distinct. Set `TolX` to `0` to obtain the results of every local solver run. Set `TolX` to a larger value to have fewer results.

Default: `1e-6`

`UseParallel`

Distribute local solver calls to multiple processors:

- `true` — Distribute the local solver calls to multiple processors.
- `false` — Cannot not run in parallel.

Default: `false`

Methods

run	Run local solver from multiple points

Copy Semantics

Value. To learn how value classes affect copy operations, see Copying Objects in the MATLAB Programming Fundamentals documentation.

Examples

Run `MultiStart` on 20 instances of a problem using the `fmincon` `sqp` algorithm:

```
opts = optimoptions(@fmincon,'Algorithm','sqp');
problem = createOptimProblem('fmincon','objective',...
 @(x) x.^2 + 4*sin(5*x),'x0',3,'lb',-5,'ub',5,'options',opts);
ms = MultiStart;
[x,f] = run(ms,problem,20)
```

Algorithms

A detailed description of the algorithm appears in "MultiStart Algorithm" on page 3-50.

See Also

`RandomStartPointSet` | `GlobalOptimSolution` | `GlobalSearch` | `createOptimProblem` | `CustomStartPointSet`

particleswarm

Particle swarm optimization

Syntax

```
x = particleswarm(fun,nvars)
x = particleswarm(fun,nvars,lb,ub)
x = particleswarm(fun,nvars,lb,ub,options)
x = particleswarm(problem)
[x,fval,exitflag,output] = particleswarm(___)
```

Description

`x = particleswarm(fun,nvars)` attempts to find a vector `x` that achieves a local minimum of `fun`. `nvars` is the dimension (number of design variables) of `fun`.

`x = particleswarm(fun,nvars,lb,ub)` defines a set of lower and upper bounds on the design variables, `x`, so that a solution is found in the range `lb` ≤ `x` ≤ `ub`.

`x = particleswarm(fun,nvars,lb,ub,options)` minimizes with the default optimization parameters replaced by values in `options`. Set `lb = []` and `ub = []` if no bounds exist.

`x = particleswarm(problem)` finds the minimum for `problem`, where `problem` is a structure.

`[x,fval,exitflag,output] = particleswarm(___)`, for any input arguments described above, returns:

- A scalar `fval`, which is the objective function value `fun(x)`
- A value `exitflag` describing the exit condition
- A structure `output` containing information about the optimization process

Examples

Minimize a Simple Function

Minimize a simple function of two variables.

Define the objective function.

```
fun = @(x)x(1)*exp(-norm(x)^2);
```

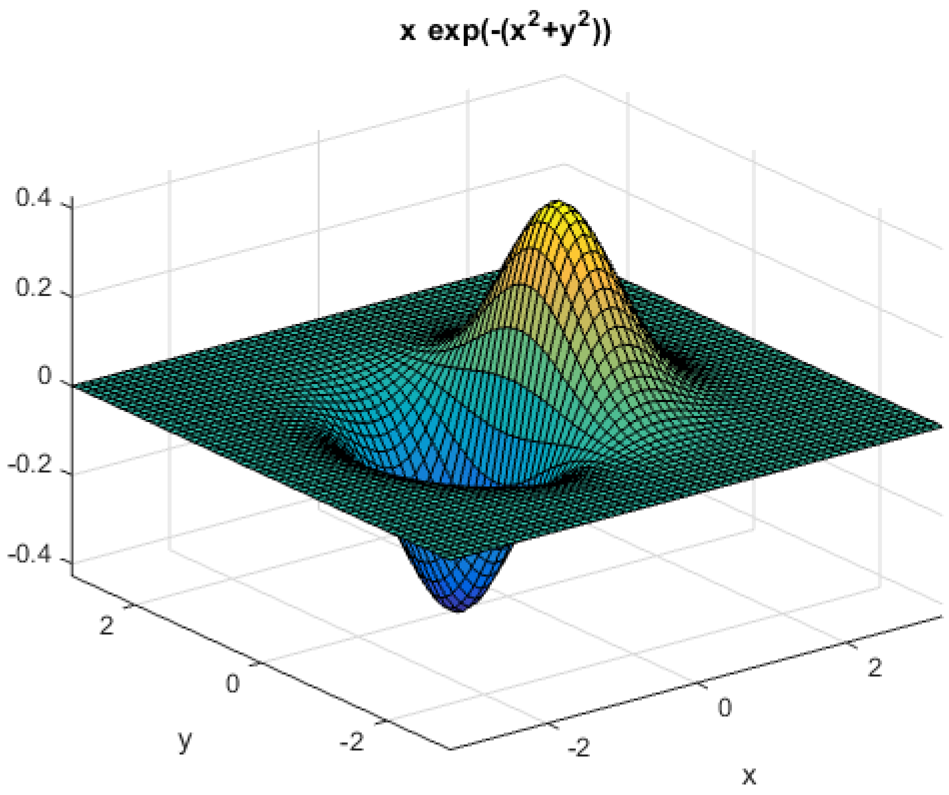
Call `particleswarm` to minimize the function.

```
rng default  % For reproducibility
x = particleswarm(fun,2)
```

```
Optimization ended: relative change in the objective value
over the last OPTIONS.StallIterLimit iterations is less than OPTIONS.TolFun.

x =

  629.4474  311.4814
```

This solution is far from the true minimum, as you see in a function plot.

```
ezsurf(@(x,y)x.*exp(-(x.^2+y.^2)))
```



Usually, it is best to set bounds. See “Minimize a Simple Function with Bounds” on page 11-53.

Minimize a Simple Function with Bounds

Minimize a simple function of two variables with bound constraints.

Define the objective function.

```
fun = @(x)x(1)*exp(-norm(x)^2);
```

Set bounds on the variables.

```
lb = [-10,-15];
```

```
ub = [15,20];
```

Call `particleswarm` to minimize the function.

```
rng default  % For reproducibility
x = particleswarm(fun,2,lb,ub)
```

```
Optimization ended: change in the objective value less than options.TolFun.

x =

   -0.7071   -0.0000
```

Minimize Using Nondefault Options

Use a larger population and a hybrid function to try to get a better solution.

Specify the objective function and bounds.

```
fun = @(x)x(1)*exp(-norm(x)^2);
lb = [-10,-15];
ub = [15,20];
```

Specify the options.

```
options = optimoptions('particleswarm','SwarmSize',100,'HybridFcn',@fmincon);
```

Call `particleswarm` to minimize the function.

```
rng default  % For reproducibility
x = particleswarm(fun,2,lb,ub,options)
```

```
Optimization ended: change in the objective value less than options.TolFun.

x =

   -0.7071   -0.0000
```

Examine the Solution Process

Return the optional output arguments to examine the solution process in more detail.

Define the problem.

```
fun = @(x)x(1)*exp(-norm(x)^2);
lb = [-10,-15];
```

```
ub = [15,20];
options = optimoptions('particleswarm','SwarmSize',50,'HybridFcn',@fmincon);
```

Call `particleswarm` with all outputs to minimize the function and get information about the solution process.

```
rng default  % For reproducibility
[x,fval,exitflag,output] = particleswarm(fun,2,lb,ub,options)
```

```
Optimization ended: change in the objective value less than options.TolFun.

x =

   -0.7071   -0.0000


fval =

   -0.4289


exitflag =

     1


output =

      rngstate: [1x1 struct]
    iterations: 43
     funccount: 2200
       message: 'Optimization ended: change in the objective value less than options..
```

- “Optimize Using Particle Swarm” on page 6-3
- “Particle Swarm Output Function” on page 6-7

Input Arguments

`fun` — Objective function

function handle

Objective function, specified as a function handle. The objective function should accept a row vector of length nvars and return a scalar value.

When the `'Vectorized'` option is `'on'`, `fun` should accept a `swarm`-by-`nvars` matrix, where `swarm` is the current population size. In this case, `fun` should return a vector the same length as `swarm` containing the fitness function values. `fun` should not assume any particular size for `swarm`, since `particleswarm` can pass a single member of a population even in a vectorized calculation.

Example: `fun = @(x)(x-[4,2]).^2`

Data Types: `function_handle`

`nvars` — Number of variables
positive integer

Number of variables, specified as a positive integer.

Example: `4`

Data Types: `double`

`lb` — Lower bounds
`[ ]` (default) | real vector or array

Lower bounds, specified as a vector or array of doubles. `lb` represents the lower bounds element-wise in `lb ≤ x ≤ ub`.

Internally, `particleswarm` converts an array `lb` to the vector `lb(:)`.

Example: `lb = [0;-Inf;4]` means `x(1) ≥ 0`, `x(3) ≥ 4`.

Data Types: `double`

`ub` — Upper bounds
`[ ]` (default) | real vector or array

Upper bounds, specified as a vector or array of doubles. `ub` represents the upper bounds element-wise in `lb ≤ x ≤ ub`.

Internally, `particleswarm` converts an array `ub` to the vector `ub(:)`.

Example: `ub = [Inf;4;10]` means `x(2) ≤ 4`, `x(3) ≤ 10`.

Data Types: `double`

`options` — Options for `particleswarm`
options created using `optimoptions`

Options for `particleswarm`, specified as the output of the `optimoptions` function.

`CreationFcn`	Handle to the function that creates the initial swarm. Default is `@pswcreationuniform`. See "Swarm Creation" on page 10-54.
`Display`	Level of display returned to the command line. • `'off'` or `'none'` displays no output. • `'final'` displays just the final output (default). • `'iter'` gives iterative display.
`DisplayInterval`	Interval for iterative display. The iterative display prints one line for every `DisplayInterval` iterations. Default is `1`.
`FunValCheck`	Check whether objective function and constraints values are valid. `'on'` displays an error when the objective function or constraints return a value that is complex, `Inf`, or `NaN`. The default, `'off'`, displays no error.
`HybridFcn`	Handle to a function that continues the optimization after `particleswarm` terminates. Possible values: • `@fmincon` • `@fminsearch` • `@fminunc` • `@patternsearch` Can also be a cell array specifying the hybrid function and its options structure, such as `{@fmincon,fminconopts}`. Default is `[]`. See "Hybrid Function" on page 10-57.
`InertiaRange`	Two-element real vector with same sign values in increasing order. Gives the lower and upper bound of the adaptive inertia. To obtain a constant (nonadaptive) inertia, set both elements of `InertiaRange` to the same value. Default is `[0.1,1.1]`. See "Particle Swarm Optimization Algorithm" on page 6-10.
`InitialSwarm`	Initial population or partial population of particles. `M`-by-`nvars` matrix, where each row represents one particle. If `M` < `SwarmSize`, then `particleswarm` creates more particles so that the total number is `SwarmSize`. If `M` > `SwarmSize`, then `particleswarm` uses the first `SwarmSize` rows.

`InitialSwarmSpan`	Initial range of particle positions that `@pswcreationuniform` creates. Can be a positive scalar or a vector with `nvars` elements, where `nvars` is the number of variables. The range for any particle component is `-InitialSwarmSpan/2,InitialSwarmSpan/2`, shifted and scaled if necessary to match any bounds. Default is `2000`.
`MaxIter`	Maximum number of iterations `particleswarm` takes. Default is `200*nvars`, where `nvars` is the number of variables.
`MaxTime`	Maximum time in seconds that `particleswarm` runs. Default is `Inf`.
`MinFractionNeighbors`	Minimum adaptive neighborhood size, a scalar from `0` to `1`. Default is `0.25`. See "Particle Swarm Optimization Algorithm" on page 6-10.
`ObjectiveLimit`	Minimum objective value, a stopping criterion. Scalar, with default `-Inf`.
`OutputFcns`	Function handle or cell array of function handles. Output functions can read iterative data, and stop the solver. Default is `[]`. See "Output Function and Plot Function" on page 10-57.
`PlotFcns`	Function handle or cell array of function handles. Plot functions can read iterative data, plot each iteration, and stop the solver. Default is `[]`. See "Output Function and Plot Function" on page 10-57.
`SelfAdjustment`	Weighting of each particle's best position when adjusting velocity. Finite scalar with default `1.49`. See "Particle Swarm Optimization Algorithm" on page 6-10.
`SocialAdjustment`	Weighting of the neighborhood's best position when adjusting velocity. Finite scalar with default `1.49`. See "Particle Swarm Optimization Algorithm" on page 6-10.
`StallIterLimit`	Positive integer with default `20`. Iterations end when the relative change in best objective function value over the last `StallIterLimit` iterations is less than `options.TolFun`.
`StallTimeLimit`	Maximum number of seconds without an improvement in the best known objective function value. Positive scalar with default `Inf`.
`SwarmSize`	Number of particles in the swarm, an integer greater than `1`. Default is `min(100,10*nvars)`, where `nvars` is the number of variables.

`TolFun`	Nonnegative scalar with default `1e-6`. Iterations end when the relative change in best objective function value over the last `StallIterLimit` iterations is less than `options.TolFun`.
`UseParallel`	Compute objective function in parallel when `true`. Default is `false`. See "Parallel or Vectorized Function Evaluation" on page 10-59.
`Vectorized`	Compute objective function in vectorized fashion when `'on'`. Default is `'off'`. See "Parallel or Vectorized Function Evaluation" on page 10-59.

`problem` — Optimization problem
structure

Optimization problem, specified as a structure with the following fields.

`solver`	`'particleswarm'`
`objective`	Function handle to the objective function, or string with the name of the objective function.
`nvars`	Number of variables in problem.
`lb`	Vector or array of lower bounds.
`ub`	Vector or array of upper bounds.
`options`	Options created by `optimoptions`.
`rngstate`	Optional state of the random number generator at the beginning of the solution process.

Data Types: `struct`

Output Arguments

`x` — Solution
real vector

Solution, returned as a real vector that minimizes the objective function subject to any bound constraints.

`fval` — Objective value
real scalar

Objective value, returned as the real scalar `fun(x)`.

`exitflag` — Algorithm stopping condition

integer

Algorithm stopping condition, returned as an integer identifying the reason the algorithm stopped. The following lists the values of `exitflag` and the corresponding reasons `particleswarm` stopped.

`1`	Relative change in the objective value over the last `options.StallIterLimit` iterations is less than `options.TolFun`.
`0`	Number of iterations exceeded `options.MaxIter`.
`-1`	Iterations stopped by output function or plot function.
`-2`	Bounds are inconsistent: for some `i`, `lb(i)` > `ub(i)`.
`-3`	Best objective function value is at or below `options.ObjectiveLimit`.
`-4`	Best objective function value did not change within `options.StallTimeLimit` seconds.
`-5`	Run time exceeded `options.MaxTime` seconds.

`output` — Solution process summary

structure

Solution process summary, returned as a structure containing information about the optimization process.

`iterations`	Number of solver iterations
`funccount`	Number of objective function evaluations.
`message`	String giving the reason the algorithm stopped.
`rngstate`	State of the default random number generator just before the algorithm started.

More About

Algorithms

For a description of the particle swarm optimization algorithm, see “Particle Swarm Optimization Algorithm” on page 6-10.

- “What Is Particle Swarm Optimization?” on page 6-2
- “Optimization Problem Setup”

See Also
`ga` | `patternsearch`

patternsearch

Find minimum of function using pattern search

Syntax

```
x = patternsearch(fun,x0)
x = patternsearch(fun,x0,A,b)
x = patternsearch(fun,x0,A,b,Aeq,beq)
x = patternsearch(fun,x0,A,b,Aeq,beq,LB,UB)
x = patternsearch(fun,x0,A,b,Aeq,beq,LB,UB,nonlcon)
x = patternsearch(fun,x0,A,b,Aeq,beq,LB,UB,nonlcon,options)
x = patternsearch(problem)
[x,fval] = patternsearch(fun,x0, ...)
[x,fval,exitflag] = patternsearch(fun,x0, ...)
[x,fval,exitflag,output] = patternsearch(fun,x0, ...)
```

Description

`patternsearch` finds the minimum of a function using a pattern search.

`x = patternsearch(fun,x0)` finds a local minimum, `x`, to the function handle `fun` that computes the values of the objective function. For details on writing `fun`, see “Compute Objective Functions” on page 2-2. `x0` is an initial point for the pattern search algorithm, a real vector.

Note To write a function with additional parameters to the independent variables that can be called by `patternsearch`, see the section on “Passing Extra Parameters” in the Optimization Toolbox documentation.

`x = patternsearch(fun,x0,A,b)` finds a local minimum `x` to the function `fun`, subject to the linear inequality constraints represented in matrix form by $Ax \le b$, see “Linear Inequality Constraints”.

If the problem has `m` linear inequality constraints and `n` variables, then

- `A` is a matrix of size `m`-by-`n`.
- `b` is a vector of length `m`.

`x = patternsearch(fun,x0,A,b,Aeq,beq)` finds a local minimum `x` to the function `fun`, starting at `x0`, and subject to the constraints

$$\begin{aligned} A * x &\le b \\ Aeq * x &= beq, \end{aligned}$$

where $Aeq * x = beq$ represents the linear equality constraints in matrix form, see "Linear Equality Constraints". If the problem has `r` linear equality constraints and `n` variables, then

- `Aeq` is a matrix of size `r`-by-`n`.
- `beq` is a vector of length `r`.

If there are no inequality constraints, pass empty matrices, `[]`, for `A` and `b`.

`x = patternsearch(fun,x0,A,b,Aeq,beq,LB,UB)` defines a set of lower and upper bounds on the design variables, `x`, so that a solution is found in the range `LB` ≤ `x` ≤ `UB`, see "Bound Constraints". If the problem has *n* variables, `LB` and `UB` are vectors of length *n*. If `LB` or `UB` is empty (or not provided), it is automatically expanded to `-Inf` or `Inf`, respectively. If there are no inequality or equality constraints, pass empty matrices for `A`, `b`, `Aeq` and `beq`.

`x = patternsearch(fun,x0,A,b,Aeq,beq,LB,UB,nonlcon)` subjects the minimization to the constraints defined in `nonlcon`, a nonlinear constraint function. The function `nonlcon` accepts `x` and returns the vectors *C* and *Ceq*, representing the nonlinear inequalities and equalities respectively. `patternsearch` minimizes `fun` such that $C(x) \le 0$ and $Ceq(x) = 0$. (Set `LB=[]` and `UB=[]` if no bounds exist.)

`x = patternsearch(fun,x0,A,b,Aeq,beq,LB,UB,nonlcon,options)` minimizes `fun` with the default optimization parameters replaced by values in `options`. The structure `options` can be created using `psoptimset`.

`x = patternsearch(problem)` finds the minimum for `problem`, where `problem` is a structure containing the following fields:

- `objective` — Objective function

- `X0` — Starting point
- `Aineq` — Matrix for linear inequality constraints
- `bineq` — Vector for linear inequality constraints
- `Aeq` — Matrix for linear equality constraints
- `beq` — Vector for linear equality constraints
- `lb` — Lower bound for `x`
- `ub` — Upper bound for `x`
- `nonlcon` — Nonlinear constraint function
- `solver` — `'patternsearch'`
- `options` — Options structure created with `psoptimset`
- `rngstate` — Optional field to reset the state of the random number generator

Create the structure `problem` by exporting a problem from the Optimization app, as described in "Importing and Exporting Your Work" in the Optimization Toolbox documentation.

Note `problem` must have all the fields as specified above.

`[x,fval] = patternsearch(fun,x0, ...)` returns the value of the objective function `fun` at the solution `x`.

`[x,fval,exitflag] = patternsearch(fun,x0, ...)` returns `exitflag`, which describes the exit condition of `patternsearch`. Possible values of `exitflag` and the corresponding conditions are

Exit Flag	Meaning
1	**Without nonlinear constraints** — Magnitude of the mesh size is less than the specified tolerance and constraint violation is less than `TolCon`.
	With nonlinear constraints — Magnitude of the *complementarity measure* (defined after this table) is less than `sqrt(TolCon)`, the subproblem is solved using a mesh finer than `TolMesh`, and the constraint violation is less than `TolCon`.
2	Change in `x` and the mesh size are both less than the specified tolerance, and the constraint violation is less than `TolCon`.

Exit Flag	Meaning
3	Change in `fval` and the mesh size are both less than the specified tolerance, and the constraint violation is less than `TolCon`.
4	Magnitude of step smaller than machine precision and the constraint violation is less than `TolCon`.
0	Maximum number of function evaluations or iterations reached.
-1	Optimization terminated by an output function or plot function.
-2	No feasible point found.

In the nonlinear constraint solver, the *complementarity measure* is the norm of the vector whose elements are $c_i\lambda_i$, where c_i is the nonlinear inequality constraint violation, and λ_i is the corresponding Lagrange multiplier.

`[x,fval,exitflag,output] = patternsearch(fun,x0, ...)` returns a structure `output` containing information about the search. The output structure contains the following fields:

- `function` — Objective function.
- `problemtype` — String describing the type of problem, one of:
  - `'unconstrained'`
  - `'boundconstraints'`
  - `'linearconstraints'`
  - `'nonlinearconstr'`
- `pollmethod` — Polling technique.
- `searchmethod` — Search technique used, if any.
- `iterations` — Total number of iterations.
- `funccount` — Total number of function evaluations.
- `meshsize` — Mesh size at `x`.
- `maxconstraint` — Maximum constraint violation, if any.
- `rngstate` — State of the MATLAB random number generator, just before the algorithm started. You can use the values in `rngstate` to reproduce the output when you use a random search method or random poll method. See “Reproduce Results” on page 5-68, which discusses the identical technique for `ga`.
- `message` — Reason why the algorithm terminated.

Note `patternsearch` does not accept functions whose inputs are of type `complex`. To solve problems involving complex data, write your functions so that they accept real vectors, by separating the real and imaginary parts.

Examples

Given the following constraints

$$\begin{bmatrix} 1 & 1 \\ -1 & 2 \\ 2 & 1 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \end{bmatrix} \le \begin{bmatrix} 2 \\ 2 \\ 3 \end{bmatrix},$$
$$x_1 \ge 0, \quad x_2 \ge 0,$$

the following code finds the minimum of the function, `lincontest6`, that is provided with your software:

```
A = [1 1; -1 2; 2 1];
b = [2; 2; 3];
lb = zeros(2,1);
[x,fval,exitflag] = patternsearch(@lincontest6,[0 0],...
                                A,b,[],[],lb)
Optimization terminated: mesh size less than
                        options.TolMesh.

x =
    0.6667    1.3333

fval =
   -8.2222

exitflag =
     1
```

More About

- "Direct Search"
- "Optimization Problem Setup"

References

[1] Audet, Charles and J. E. Dennis Jr. “Analysis of Generalized Pattern Searches.” *SIAM Journal on Optimization*, Volume 13, Number 3, 2003, pp. 889–903.

[2] Conn, A. R., N. I. M. Gould, and Ph. L. Toint. “A Globally Convergent Augmented Lagrangian Barrier Algorithm for Optimization with General Inequality Constraints and Simple Bounds.” *Mathematics of Computation*, Volume 66, Number 217, 1997, pp. 261–288.

[3] Abramson, Mark A. *Pattern Search Filter Algorithms for Mixed Variable General Constrained Optimization Problems*. Ph.D. Thesis, Department of Computational and Applied Mathematics, Rice University, August 2002.

[4] Abramson, Mark A., Charles Audet, J. E. Dennis, Jr., and Sebastien Le Digabel. “ORTHOMADS: A deterministic MADS instance with orthogonal directions.” *SIAM Journal on Optimization*, Volume 20, Number 2, 2009, pp. 948–966.

[5] Kolda, Tamara G., Robert Michael Lewis, and Virginia Torczon. “Optimization by direct search: new perspectives on some classical and modern methods.” *SIAM Review*, Volume 45, Issue 3, 2003, pp. 385–482.

[6] Kolda, Tamara G., Robert Michael Lewis, and Virginia Torczon. “A generating set direct search augmented Lagrangian algorithm for optimization with a combination of general and linear constraints.” Technical Report SAND2006-5315, Sandia National Laboratories, August 2006.

[7] Lewis, Robert Michael, Anne Shepherd, and Virginia Torczon. “Implementing generating set search methods for linearly constrained minimization.” *SIAM Journal on Scientific Computing*, Volume 29, Issue 6, 2007, pp. 2507–2530.

See Also

`ga` | `optimtool` | `particleswarm` | `psoptimset`

psoptimget

Obtain values of pattern search options structure

Syntax

```
val = psoptimget(options,'name')
val = psoptimget(options,'name',default)
```

Description

`val = psoptimget(options,'name')` returns the value of the parameter `name` from the pattern search options structure `options`. `psoptimget(options,'name')` returns an empty matrix `[]` if the value of `name` is not specified in `options`. It is only necessary to type enough leading characters of `name` to uniquely identify it. `psoptimget` ignores case in parameter names.

`val = psoptimget(options,'name',default)` returns the value of the parameter `name` from the pattern search options structure `options`, but returns `default` if the parameter is not specified (as in `[]`) in `options`.

Examples

```
val = psoptimget(opts,'TolX',1e-4);
```

returns `val = 1e-4` if the `TolX` property is not specified in `opts`.

More About

- "Pattern Search Options" on page 10-9

See Also

`psoptimset` | `patternsearch`

psoptimset

Create pattern search options structure

Syntax

```
psoptimset
options = psoptimset
options = psoptimset(@patternsearch)
options = psoptimset('param1',value1,'param2',value2,...)
options = psoptimset(oldopts,'param1',value1,...)
options = psoptimset(oldopts,newopts)
```

Description

`psoptimset` with no input or output arguments displays a complete list of parameters with their valid values.

`options = psoptimset` (with no input arguments) creates a structure called `options` that contains the options, or *parameters*, for `patternsearch`, and sets parameters to `[]`, indicating `patternsearch` uses the default values.

`options = psoptimset(@patternsearch)` creates a structure called `options` that contains the default values for `patternsearch`.

`options = psoptimset('param1',value1,'param2',value2,...)` creates a structure `options` and sets the value of `'param1'` to `value1`, `'param2'` to `value2`, and so on. Any unspecified parameters are set to their default values. It is sufficient to type only enough leading characters to define the parameter name uniquely. Case is ignored for parameter names.

`options = psoptimset(oldopts,'param1',value1,...)` creates a copy of `oldopts`, modifying the specified parameters with the specified values.

`options = psoptimset(oldopts,newopts)` combines an existing options structure, `oldopts`, with a new options structure, `newopts`. Any parameters in `newopts` with nonempty values overwrite the corresponding old parameters in `oldopts`.

Options

The following table lists the options you can set with `psoptimset`. See “Pattern Search Options” on page 10-9 for a complete description of the options and their values. Values in `{}` denote the default value. You can also view the optimization parameters and defaults by typing `psoptimset` at the command line.

Option	Description	Values
`Cache`	With `Cache` set to `'on'`, `patternsearch` keeps a history of the mesh points it polls and does not poll points close to them again at subsequent iterations. Use this option if `patternsearch` runs slowly because it is taking a long time to compute the objective function. If the objective function is stochastic, it is advised not to use this option.	`'on'` | `{'off'}`
`CacheSize`	Size of the history	Positive scalar | `{1e4}`
`CacheTol`	Positive scalar specifying how close the current mesh point must be to a point in the history in order for `patternsearch` to avoid polling it. Use if `'Cache'` option is set to `'on'`.	Positive scalar | `{eps}`
`CompletePoll`	Complete poll around current iterate	`'on'` | `{'off'}`
`CompleteSearch`	Complete search around current iterate when the search method is a poll method	`'on'` | `{'off'}`
`Display`	Level of display	`'off'` | `'iter'` | `'diagnose'` | `{'final'}`

Option	Description	Values
`InitialMeshSize`	Initial mesh size for pattern algorithm	Positive scalar | `{1.0}`
`InitialPenalty`	Initial value of the penalty parameter	Positive scalar | `{10}`
`MaxFunEvals`	Maximum number of objective function evaluations	Positive integer | `{2000*numberOfVariables}`
`MaxIter`	Maximum number of iterations	Positive integer | `{100*numberOfVariables}`
`MaxMeshSize`	Maximum mesh size used in a poll/search step	Positive scalar | `{Inf}`
`MeshAccelerator`	Accelerate convergence near a minimum	`'on'` | `{'off'}`
`MeshContraction`	Mesh contraction factor, used when iteration is unsuccessful	Positive scalar | `{0.5}`
`MeshExpansion`	Mesh expansion factor, expands mesh when iteration is successful	Positive scalar | `{2.0}`
`MeshRotate`	Rotate the pattern before declaring a point to be optimum	`'off'` | `{'on'}`
`OutputFcns`	Specifies a user-defined function that an optimization function calls at each iteration	Function handle or cell array of function handles | `{[]}`
`PenaltyFactor`	Penalty update parameter	Positive scalar| `{100}`
`PlotFcns`	Specifies plots of output from the pattern search	`@psplotbestf` | `@psplotmeshsize` | `@psplotfuncount` | `@psplotbestx` | `{[]}`
`PlotInterval`	Specifies that plot functions will be called at every interval	`{1}`
`PollingOrder`	Order of poll directions in pattern search	`'Random'` | `'Success'` | `{'Consecutive'}`

Option	Description	Values
`PollMethod`	Polling strategy used in pattern search	`{'GPSPositiveBasis2N'} | 'GPSPositiveBasisNp1'| 'GSSPositiveBasis2N'| 'GSSPositiveBasisNp1'| 'MADSPositiveBasis2N'| 'MADSPositiveBasisNp1'`
`ScaleMesh`	Automatic scaling of variables	`{'on'} | 'off'`
`SearchMethod`	Type of search used in pattern search	`@GPSPositiveBasis2N | @GPSPositiveBasisNp1 | @GSSPositiveBasis2N | @GSSPositiveBasisNp1 | @MADSPositiveBasis2N | @MADSPositiveBasisNp1 | @searchga | @searchlhs | @searchneldermead | {[]}`
`TimeLimit`	Total time (in seconds) allowed for optimization	Positive scalar | `{Inf}`
`TolBind`	Binding tolerance	Positive scalar | `{1e-3}`
`TolCon`	Tolerance on constraints	Positive scalar | `{1e-6}`
`TolFun`	Tolerance on function, stop if both the change in function value and the mesh size are less than `TolFun`	Positive scalar | `{1e-6}`
`TolMesh`	Tolerance on mesh size	Positive scalar | `{1e-6}`
`TolX`	Tolerance on variable, stop if both the change in position and the mesh size are less than `TolX`	Positive scalar | `{1e-6}`

Option	Description	Values
`UseParallel`	Compute objective and nonlinear constraint functions in parallel, see "Vectorize and Parallel Options (User Function Evaluation)" on page 10-24 and "How to Use Parallel Processing" on page 9-12.	`true` | `{false}`
`Vectorized`	Specifies whether functions are vectorized, see "Vectorize and Parallel Options (User Function Evaluation)" on page 10-24 and "Vectorize the Objective and Constraint Functions" on page 4-80	`'on'` | `{'off'}`

See Also

`patternsearch` | `psoptimget`

run

Class: GlobalSearch

Find global minimum

Syntax

```
x = run(gs,problem)
[x,fval] = run(gs,problem)
[x,fval,exitflag] = run(gs,problem)
[x,fval,exitflag,output] = run(gs,problem)
[x,fval,exitflag,output,solutions] = run(gs,problem)
```

Description

`x = run(gs,problem)` finds a point `x` that solves the optimization problem described in the `problem` structure.

`[x,fval] = run(gs,problem)` returns the value of the objective function in `problem` at the point `x`.

`[x,fval,exitflag] = run(gs,problem)` returns an exit flag describing the results of the multiple local searches.

`[x,fval,exitflag,output] = run(gs,problem)` returns an output structure describing the iterations of the run.

`[x,fval,exitflag,output,solutions] = run(gs,problem)` returns a vector of solutions containing the distinct local minima found during the run.

Input Arguments

gs

A `GlobalSearch` object.

`problem`

Problem structure. Create `problem` with `createOptimProblem` or by exporting a problem structure from the Optimization app. `problem` must contain at least the following fields:

- `solver`
- `objective`
- `x0`
- `options` — Both `createOptimProblem` and the Optimization app always include an `options` field in the `problem` structure.

Output Arguments

`x`

Minimizing point of the objective function.

`fval`

Objective function value at the minimizer `x`.

`exitflag`

Describes the results of the multiple local searches. Values are:

2	At least one local minimum found. Some runs of the local solver converged.
1	At least one local minimum found. All runs of the local solver converged.
0	No local minimum found. Local solver called at least once, and at least one local solver exceeded the `MaxIter` or `MaxFunEvals` tolerances.
-1	One or more local solver runs stopped by the local solver output or plot function.
-2	No feasible local minimum found.
-5	`MaxTime` limit exceeded.

`-8`	No solution found. All runs had local solver exit flag `-2` or smaller, not all equal `-2`.
`-10`	Failures encountered in user-provided functions.

`output`

A structure describing the iterations of the run. Fields in the structure:

`funcCount`	Number of function evaluations.
`localSolverIncomplete`	Number of local solver runs with `0` exit flag.
`localSolverNoSolution`	Number of local solver runs with negative exit flag.
`localSolverSuccess`	Number of local solver runs with positive exit flag.
`localSolverTotal`	Total number of local solver runs.
`message`	Exit message.

`solutions`

A vector of `GlobalOptimSolution` objects containing the distinct local solutions found during the run. The vector is sorted by objective function value; the first element is best (smallest value). The object contains:

`X`	Solution point returned by the local solver.
`Fval`	Objective function value at the solution.
`Exitflag`	Integer describing the result of the local solver run.
`Output`	Output structure returned by the local solver.
`X0`	Cell array of start points that led to the solution.

Examples

Use a default `GlobalSearch` object to solve the six-hump camel back problem (see "Run the Solver" on page 3-20):

```
gs = GlobalSearch;
sixmin = @(x)(4*x(1)^2 - 2.1*x(1)^4 + x(1)^6/3 ...
    + x(1)*x(2) - 4*x(2)^2 + 4*x(2)^4);
problem = createOptimProblem('fmincon','x0',[-1,2],...
    'objective',sixmin,'lb',[-3,-3],'ub',[3,3]);
[xmin,fmin,flag,outpt,allmins] = run(gs,problem);
```

Algorithms

A detailed description of the algorithm appears in “GlobalSearch Algorithm” on page 3-46. Ugray et al. [1] describes both the algorithm and the scatter-search method of generating trial points.

References

[1] Ugray, Zsolt, Leon Lasdon, John Plummer, Fred Glover, James Kelly, and Rafael Martí. *Scatter Search and Local NLP Solvers: A Multistart Framework for Global Optimization*. INFORMS Journal on Computing, Vol. 19, No. 3, 2007, pp. 328–340.

See Also

`GlobalOptimSolution` | `GlobalSearch` | `createOptimProblem`

run

Class: MultiStart

Run local solver from multiple points

Syntax

```
x = run(ms,problem,k)
x = run(ms,problem,startpts)
[x,fval] = run(...)
[x,fval,exitflag] = run(...)
[x,fval,exitflag,output] = run(...)
[x,fval,exitflag,output,solutions] = run(...)
```

Description

`x = run(ms,problem,k)` runs the `ms` `MultiStart` object on the optimization problem specified in `problem` for a total of `k` runs. `x` is the point where the lowest function value was found. For the `lsqcurvefit` and `lsqnonlin` solvers, `MultiStart` minimizes the sum of squares at `x`, also known as the residual.

`x = run(ms,problem,startpts)` runs the `ms` `MultiStart` object on the optimization problem specified in `problem` using the start points described in `startpts`.

`[x,fval] = run(...)` returns the lowest objective function value `fval` at `x`. For the `lsqcurvefit` and `lsqnonlin` solvers, `fval` contains the residual.

`[x,fval,exitflag] = run(...)` returns an exit flag describing the return condition.

`[x,fval,exitflag,output] = run(...)` returns an output structure containing statistics of the run.

`[x,fval,exitflag,output,solutions] = run(...)` returns a vector of solutions containing the distinct local minima found during the run.

Input Arguments

`ms`

`MultiStart` object.

`problem`

Problem structure. Create `problem` with `createOptimProblem` or by exporting a problem structure from the Optimization app. `problem` must contain the following fields:

- `solver`
- `objective`
- `x0`
- `options` — Both `createOptimProblem` and the Optimization app always include an `options` field in the `problem` structure.

`k`

Number of start points to run. `MultiStart` generates `k - 1` start points using the same algorithm as `list` for a `RandomStartPointSet` object. `MultiStart` also uses the `x0` point from the `problem` structure.

`startpts`

A `CustomStartPointSet` or `RandomStartPointSet` object. `startpts` can also be a cell array of these objects.

Output Arguments

`x`

Point at which the objective function attained the lowest value during the run. For `lsqcurvefit` and `lsqnonlin`, the objective function is the sum of squares, also known as the residual.

`fval`

Smallest objective function value attained during the run. For `lsqcurvefit` and `lsqnonlin`, the objective function is the sum of squares, also known as the residual.

exitflag

Integer describing the return condition:

`2`	At least one local minimum found. Some runs of the local solver converged.
`1`	At least one local minimum found. All runs of the local solver converged.
`0`	No local minimum found. Local solver called at least once, and at least one local solver exceeded the `MaxIter` or `MaxFunEvals` tolerances.
`-1`	One or more local solver runs stopped by the local solver output or plot function.
`-2`	No feasible local minimum found.
`-5`	`MaxTime` limit exceeded.
`-8`	No solution found. All runs had local solver exit flag `-2` or smaller, not all equal `-2`.
`-10`	Failures encountered in user-provided functions.

output

Structure containing statistics of the optimization. Fields in the structure:

`funcCount`	Number of function evaluations.
`localSolverIncomplete`	Number of local solver runs with `0` exit flag.
`localSolverNoSolution`	Number of local solver runs with negative exit flag.
`localSolverSuccess`	Number of local solver runs with positive exit flag.
`localSolverTotal`	Total number of local solver runs.
`message`	Exit message.

solutions

A vector of `GlobalOptimSolution` objects containing the distinct local solutions found during the run. The vector is sorted by objective function value; the first element is best (smallest value). The object contains:

`X`	Solution point returned by the local solver.

`Fval`	Objective function value at the solution.
`Exitflag`	Integer describing the result of the local solver run.
`Output`	Output structure returned by the local solver.
`X0`	Cell array of start points that led to the solution.

Examples

Use a default `MultiStart` object to solve the six-hump camel back problem (see "Run the Solver" on page 3-20):

```
ms = MultiStart;
sixmin = @(x)(4*x(1)^2 - 2.1*x(1)^4 + x(1)^6/3 ...
    + x(1)*x(2) - 4*x(2)^2 + 4*x(2)^4);
problem = createOptimProblem('fmincon','x0',[-1,2],...
    'objective',sixmin,'lb',[-3,-3],'ub',[3,3]);
[xmin,fmin,flag,outpt,allmins] = run(ms,problem,30);
```

Algorithms

A detailed description of the algorithm appears in "MultiStart Algorithm" on page 3-50.

See Also

`CustomStartPointSet` | `GlobalOptimSolution` | `MultiStart` | `createOptimProblem` | `RandomStartPointSet`

RandomStartPointSet class

Random start points

Description

Describes how to generate a set of pseudorandom points for use with `MultiStart`. A `RandomStartPointSet` object does not contain points. It contains parameters used in generating the points when `MultiStart` runs, or by the `list` method.

Construction

`*RS* = RandomStartPointSet` constructs a default `RandomStartPointSet` object.

`*RS* = RandomStartPointSet('*PropertyName*',*PropertyValue*,...)` constructs the object using options, specified as property name and value pairs.

`*RS* = RandomStartPointSet(*OLDRS*,'*PropertyName*',*PropertyValue*,...)` creates a copy of the *OLDRS* `RandomStartPointSet` object, with the named properties altered with the specified values.

Properties

`ArtificialBound`

Absolute value of default bounds to use for unbounded problems (positive scalar).

If a component has no bounds, `list` uses a lower bound of `-ArtificialBound`, and an upper bound of `ArtificialBound`.

If a component has a lower bound `lb`, but no upper bound, `list` uses an upper bound of `lb + 2*ArtificialBound`. Similarly, if a component has an upper bound `ub`, but no lower bound, `list` uses a lower bound of `ub - 2*ArtificialBound`.

Default: `1000`

`NumStartPoints`

Number of start points to generate (positive integer)

Default: `10`

Methods

list	Generate start points

Copy Semantics

Value. To learn how value classes affect copy operations, see Copying Objects in the MATLAB Programming Fundamentals documentation.

Examples

Create a `RandomStartPointSet` object for 40 points, and use `list` to generate a point matrix for a seven-dimensional problem:

```
rs = RandomStartPointSet('NumStartPoints',40); % 40 points
problem = createOptimProblem('fminunc','x0',ones(7,1),...
    'objective',@rosenbrock);
ptmatrix = list(rs,problem); % 'list' generates the matrix
```

See Also

`MultiStart` | `list` | `CustomStartPointSet`

How To

- Class Attributes
- Property Attributes

saoptimget

Values of simulated annealing options structure

Syntax

```
val = saoptimget(options, 'name')
val = saoptimget(options, 'name', default)
```

Description

`val = saoptimget(options, 'name')` returns the value of the parameter `name` from the simulated annealing options structure `options`. `saoptimget(options, 'name')` returns an empty matrix `[]` if the value of `name` is not specified in `options`. It is only necessary to type enough leading characters of `name` to uniquely identify the parameter. `saoptimget` ignores case in parameter names.

`val = saoptimget(options, 'name', default)` returns the `'name'` parameter, but returns the default value if the `'name'` parameter is not specified (or is `[]`) in `options`.

Examples

```
opts = saoptimset('TolFun',1e-4);
val = saoptimget(opts,'TolFun');
```

returns `val = 1e-4` for `TolFun`.

More About

- "Simulated Annealing Options" on page 10-62

See Also

`saoptimset` | `simulannealbnd`

saoptimset

Create simulated annealing options structure

Syntax

```
saoptimset
options = saoptimset
options = saoptimset('param1',value1,'param2',value2,...)
options = saoptimset(oldopts,'param1',value1,...)
options = saoptimset(oldopts,newopts)
options = saoptimset('simulannealbnd')
```

Description

`saoptimset` with no input or output arguments displays a complete list of parameters with their valid values.

`options = saoptimset` (with no input arguments) creates a structure called `options` that contains the options, or *parameters*, for the simulated annealing algorithm, with all parameters set to `[]`.

`options = saoptimset('param1',value1,'param2',value2,...)` creates a structure `options` and sets the value of `'param1'` to `value1`, `'param2'` to `value2`, and so on. Any unspecified parameters are set to `[]`. It is sufficient to type only enough leading characters to define the parameter name uniquely. Case is ignored for parameter names. Note that for string values, correct case and the complete string are required.

`options = saoptimset(oldopts,'param1',value1,...)` creates a copy of `oldopts`, modifying the specified parameters with the specified values.

`options = saoptimset(oldopts,newopts)` combines an existing options structure, `oldopts`, with a new options structure, `newopts`. Any parameters in `newopts` with nonempty values overwrite the corresponding old parameters in `oldopts`.

`options = saoptimset('simulannealbnd')` creates an options structure with all the parameter names and default values relevant to `'simulannealbnd'`. For example,

```
saoptimset('simulannealbnd')
```

```
ans = 
          AnnealingFcn: @annealingfast
        TemperatureFcn: @temperatureexp
         AcceptanceFcn: @acceptancesa
                TolFun: 1.0000e-006
        StallIterLimit: '500*numberofvariables'
           MaxFunEvals: '3000*numberofvariables'
             TimeLimit: Inf
               MaxIter: Inf
        ObjectiveLimit: -Inf
               Display: 'final'
       DisplayInterval: 10
             HybridFcn: []
        HybridInterval: 'end'
              PlotFcns: []
          PlotInterval: 1
            OutputFcns: []
    InitialTemperature: 100
      ReannealInterval: 100
              DataType: 'double'
```

Options

The following table lists the options you can set with `saoptimset`. See "Simulated Annealing Options" on page 10-62 for a complete description of these options and their values. Values in `{}` denote the default value. You can also view the options parameters by typing `saoptimset` at the command line.

Option	Description	Values
`AcceptanceFcn`	Handle to the function the algorithm uses to determine if a new point is accepted	Function handle |`{@acceptancesa}`
`AnnealingFcn`	Handle to the function the algorithm uses to generate new points	Function handle | `@annealingboltz` | `{@annealingfast}`
`DataType`	Type of decision variable	`'custom'` | `{'double'}`
`Display`	Level of display	`'off'` | `'iter'` | `'diagnose'` | `{'final'}`
`DisplayInterval`	Interval for iterative display	Positive integer | `{10}`

Option	Description	Values
HybridFcn	Automatically run HybridFcn (another optimization function) during or at the end of iterations of the solver	@fminsearch | @patternsearch | @fminunc | @fmincon | {[]} or 1-by-2 cell array | {@solver, hybridoptions}, where solver = fminsearch, patternsearch, fminunc, or fmincon {[]}
HybridInterval	Interval (if not 'end' or 'never') at which HybridFcn is called	Positive integer | 'never' | {'end'}
InitialTemperature	Initial value of temperature	Positive scalar | positive vector | {100}
MaxFunEvals	Maximum number of objective function evaluations allowed	Positive integer | {3000*numberOfVariables}
MaxIter	Maximum number of iterations allowed	Positive integer | {Inf}
ObjectiveLimit	Minimum objective function value desired	Scalar | {-Inf}
OutputFcns	Function(s) get(s) iterative data and can change options at run time	Function handle or cell array of function handles | {[]}
PlotFcns	Plot function(s) called during iterations	Function handle or cell array of function handles | @saplotbestf | @saplotbestx | @saplotf | @saplotstopping | @saplottemperature | {[]}
PlotInterval	Plot functions are called at every interval	Positive integer | {1}
ReannealInterval	Reannealing interval	Positive integer | {100}

Option	Description	Values
`StallIterLimit`	Number of iterations over which average change in fitness function value at current point is less than `options.TolFun`	Positive integer | `{500*numberOfVariables}`
`TemperatureFcn`	Function used to update temperature schedule	Function handle | `@temperatureboltz` | `@temperaturefast` | `{@temperatureexp}`
`TimeLimit`	The algorithm stops after running for `TimeLimit` seconds	Positive scalar | `{Inf}`
`TolFun`	Termination tolerance on function value	Positive scalar | `{1e-6}`

More About

- "Simulated Annealing Options" on page 10-62

See Also

`saoptimget` | `simulannealbnd`

simulannealbnd

Find minimum of function using simulated annealing algorithm

Syntax

```
x = simulannealbnd(fun,x0)
x = simulannealbnd(fun,x0,lb,ub)
x = simulannealbnd(fun,x0,lb,ub,options)
x = simulannealbnd(problem)
[x,fval] = simulannealbnd(...)
[x,fval,exitflag] = simulannealbnd(...)
[x,fval,exitflag,output] = simulannealbnd(fun,...)
```

Description

`x = simulannealbnd(fun,x0)` starts at `x0` and finds a local minimum `x` to the objective function specified by the function handle `fun`. The objective function accepts input `x` and returns a scalar function value evaluated at `x`. `x0` may be a scalar or a vector.

`x = simulannealbnd(fun,x0,lb,ub)` defines a set of lower and upper bounds on the design variables, `x`, so that a solution is found in the range `lb` ≤ `x` ≤ `ub`. Use empty matrices for `lb` and `ub` if no bounds exist. Set `lb(i)` to `-Inf` if `x(i)` is unbounded below; set `ub(i)` to `Inf` if `x(i)` is unbounded above.

`x = simulannealbnd(fun,x0,lb,ub,options)` minimizes with the default optimization parameters replaced by values in the structure `options`, which can be created using the `saoptimset` function. See the `saoptimset` reference page for details.

`x = simulannealbnd(problem)` finds the minimum for `problem`, where `problem` is a structure containing the following fields:

`objective`	Objective function
`x0`	Initial point of the search
`lb`	Lower bound on `x`

`ub`	Upper bound on `x`
`rngstate`	Optional field to reset the state of the random number generator
`solver`	`'simulannealbnd'`
`options`	Options structure created using `saoptimset`

Create the structure `problem` by exporting a problem from Optimization app, as described in "Importing and Exporting Your Work" in the Optimization Toolbox documentation.

`[x,fval] = simulannealbnd(...)` returns `fval`, the value of the objective function at `x`.

`[x,fval,exitflag] = simulannealbnd(...)` returns `exitflag`, an integer identifying the reason the algorithm terminated. The following lists the values of `exitflag` and the corresponding reasons the algorithm terminated:

- `1` — Average change in the value of the objective function over `options.StallIterLimit` iterations is less than `options.TolFun`.
- `5` — `options.ObjectiveLimit` limit reached.
- `0` — Maximum number of function evaluations or iterations exceeded.
- `-1` — Optimization terminated by an output function or plot function.
- `-2` — No feasible point found.
- `-5` — Time limit exceeded.

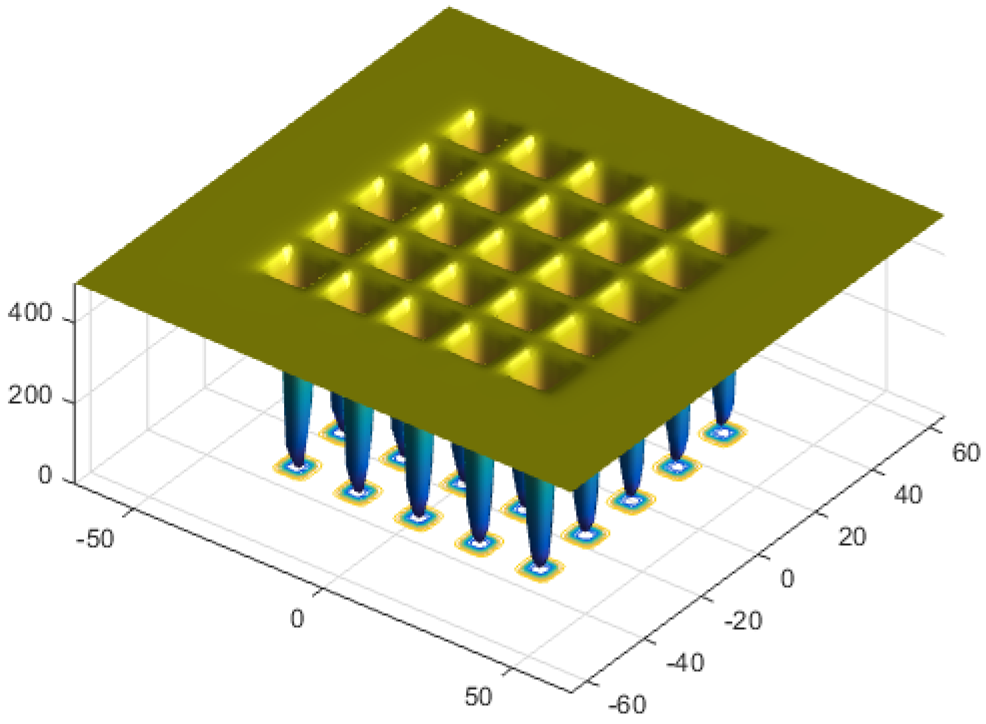
`[x,fval,exitflag,output] = simulannealbnd(fun,...)` returns `output`, a structure that contains information about the problem and the performance of the algorithm. The `output` structure contains the following fields:

- `problemtype` — Type of problem: unconstrained or bound constrained.
- `iterations` — The number of iterations computed.
- `funccount` — The number of evaluations of the objective function.
- `message` — The reason the algorithm terminated.
- `temperature` — Temperature when the solver terminated.
- `totaltime` — Total time for the solver to run.

- `rngstate` — State of the MATLAB random number generator, just before the algorithm started. You can use the values in `rngstate` to reproduce the output of `simulannealbnd`. See "Reproduce Your Results" on page 7-18.

Examples

Minimization of De Jong's fifth function, a two-dimensional function with many local minima. Enter the command `dejong5fcn` to generate the following plot.



```
x0 = [0 0];
[x,fval] = simulannealbnd(@dejong5fcn,x0)
```

```
Optimization terminated: change in best function value
                         less than options.TolFun.

x =
    0.0392  -31.9700

fval =
    2.9821
```

Minimization of De Jong's fifth function subject to lower and upper bounds:

```
x0 = [0 0];
lb = [-64 -64];
ub = [64 64];
[x,fval] = simulannealbnd(@dejong5fcn,x0,lb,ub)

Optimization terminated: change in best function value
                         less than options.TolFun.

x =
  -31.9652  -32.0286

fval =
    0.9980
```

The objective can also be an anonymous function:

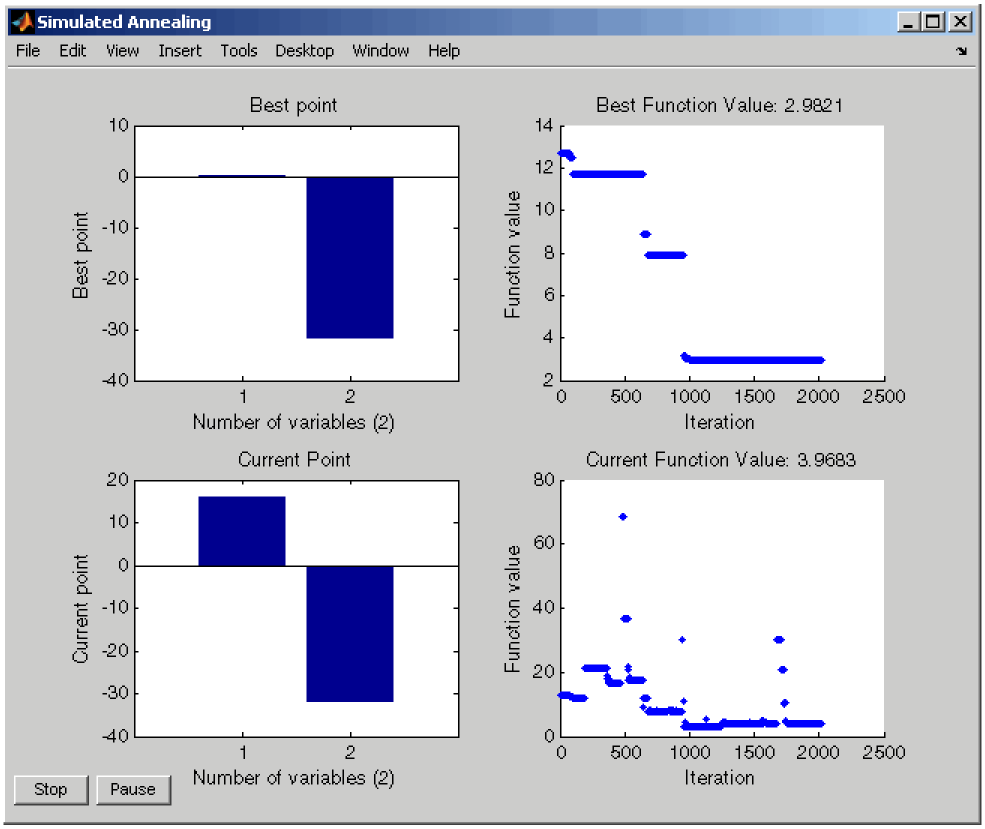
```
fun =  @(x) 3*sin(x(1))+exp(x(2));
x = simulannealbnd(fun,[1;1],[0 0])

Optimization terminated: change in best function value
                         less than options.TolFun.

x =
  457.1045
    0.0000
```

Minimization of De Jong's fifth function while displaying plots:

```
x0 = [0 0];
options = saoptimset('PlotFcns',{@saplotbestx,...
               @saplotbestf,@saplotx,@saplotf});
simulannealbnd(@dejong5fcn,x0,[],[],options)

Optimization terminated: change in best function value
```

```
                          less than options.TolFun.

ans =
    0.0230   -31.9806
```

The plots displayed are shown below.



See Also

`ga` | `patternsearch` | `saoptimget` | `saoptimset`